LOW-COST AND EFFICIENT ARCHITECTURAL SUPPORT FOR CORRECTNESS AND PERFORMANCE DEBUGGING

A Thesis
Presented to
The Academic Faculty

by

Guru Prasadh V. Venkataramani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2009

LOW-COST AND EFFICIENT ARCHITECTURAL SUPPORT FOR CORRECTNESS AND PERFORMANCE DEBUGGING

Approved by:

Prof. Milos Prvulovic, Advisor
School of Computer Science
Georgia Institute of Technology

Prof. Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Gabriel H. Loh
School of Computer Science
Georgia Institute of Technology

Dr. Christopher J. Hughes
Throughput Computing Lab
Intel Corporation

Prof. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Date Approved: 1 July 2009

Dedicated to Amma, Appa and Sudha

ACKNOWLEDGEMENTS

My first thanks are due to my advisor, Prof. Milos Prvulovic. He has been a great guide and a constant source of energy by motivating me to perform better at all times. I have always taken the liberty to stop by his office and discuss my ideas. On all such occasions, he made me feel very welcome and has given me forthright advice when I needed them the most. Our technical conversations were always spiced up by his funny thoughts on almost everything we discussed and usually ended up being intellectual and friendly at the same time.

I am also fortunate to have PhD examining committee members who are not just people assembled to read my thesis – they have truly taken part in my successes. Prof. Gabriel Loh inspired me to work harder, resist mediocrity, and be lively at all times. Prof. Hsien-Hsin Lee is an information repository who could easily involve me in conversations on any topic. He helped me with his great advice on numerous occasions. Dr. Christopher Hughes, who was my mentor during my internship at Intel Corporation, has been a constant source of strength and encouragement. He has been extremely generous with his time to provide constructive feedback for all my ideas. Even though my internship lasted for just three months, we collaborated for almost three years and my work on performance debugging was a result of his encouragement. Prof. Hyesoon Kim, who I got to know in the latter part of my PhD years, has inspired me to be diligent and patient. She has given me useful tips on how to prepare job application materials and stand out as a promising candidate in the market.

My friends also played crucial roles in my life as a PhD student. My good friend and roommate, Vijay has been very supportive all these five years. Ioannis frequently helped me to hone my research ideas and presentation slides, Samantika and Kiran would engage

me in lively discussions, Jungju patiently listened to all my practice talks, Chenyu and Yuejian helped me clarify several technical doubts, Lal advised me to stay firm and calm on several stressful situations. Lenin, Meenakshi Sundaram and Vishwanath have helped me on various occasions. I would also like to thank Alan, Becky and Susie for their prompt administrative support.

Finally, my family has been by my side in all my endeavors. My mother constantly encouraged me to have higher goals and work harder, my father taught me how to be honest and organized, and my sister was my role-model for unbounded optimism. They had faith in whatever I ventured to do and I wouldn't be here without their love and support. My family saw the best in me and I am forever grateful to them.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES xiii

LIST OF FIGURES xviii

SUMMARY xix

I INTRODUCTION 1

1.1 Motivation 1

1.2 Architectural Support for Debugging 3

1.3 Overview 6

1.4 Scope of this Dissertation 7

II BACKGROUND 8

2.1 Memory Debugging 8

2.2 Tainting 10

2.3 Performance Debugging 12

2.4 False Sharing and its Implications on Scalability 16

2.4.1 Definition 17

2.4.2 Real-World examples 18

2.4.3 Summary 21

III MEMTRACKER: MEMORY ACCESS MONITOR AND DEBUGGER 22

3.1 Motivation 22

3.2 Overview 23

3.2.1 HeapData: an Example Memory Access Checker 23

3.2.2 Functionality 25

3.2.3 Events 26

3.2.4 Setup for the HeapData Checker 28

3.2.5 Dealing with Sub-Word Accesses 29

3.2.6 Event Masking 31

3.2.7 Support for multiple checkers 31

3.3 Implementation 34

3.3.1 Programmable State Transition Table (PSTT) 34

3.3.2 Finding State Information 35

3.3.3 Caching State Information 36

3.3.4 Processor Modifications for MemTracker 37

3.3.5 Filtering of Silent State Writes 40

3.3.6 Multiprocessor Consistency Issues 40

3.3.7 Setup and Context-Switching 42

3.4 Evaluation Setup 43
3.4.1 Memory Problem Detectors 43
3.4.2 Benchmark Applications 46
3.4.3 Simulation Environment and Configuration 46
3.5 Evaluation 47
3.5.1 Effect of L1 Caching Approaches 47
3.5.2 Performance with Different Checkers 49
3.5.3 Sensitivity Analysis 51
3.5.4 Comparison with Prior Mechanisms 52
3.5.5 Latency, area and power overheads 55
3.5.6 Validation of Access Checking Functionality 56
3.6 Related Work 57
3.7 Summary 60
IV FLEXITAINT: TAINT PROPAGATION ACCELERATOR 62
4.1 Motivation 62
4.2 Overview 63
4.2.1 Storing Taint Information 63
4.2.2 Processing Taint Information 65
4.2.3 Programmable Taint Propagation 68

4.2.4 Taint Manipulation Instructions 69
4.2.5 Fast Common-Case Taint Propagation 71
4.3 Implementation 72
4.3.1 Multiprocessor Consistency Issues 74
4.3.2 Initialization and OS Interaction 75
4.4 Evaluation Setup 76
4.4.1 Taint Propagation Schemes 77
4.4.2 Benchmark Applications 78
4.4.3 Simulator and Configuration 79
4.5 Evaluation 79
4.5.1 Performance with different taint propagation policies 79
4.5.2 Effect of common-case optimizations 81
4.5.3 Effect of silent taint writes 82
4.5.4 Sensitivity Analysis 83
4.5.5 Effect of limited programmability 84
4.5.6 Validation 86
4.6 Related Work 86
4.7 Summary 88
V CACHEDOC: COMPREHENSIVE CACHE MISS CLASSIFIER 90

5.1 Motivation 90

5.2 Classification of Replacement Misses 91

5.2.1 Ideal Conflict Miss Detector (I-CMD) 91

5.2.2 Practical Conflict Miss Detector 92

5.2.3 Implementation 93

5.2.4 Evaluation 97

5.3 Classification of Coherence Misses 104

5.3.1 Identification of Coherence Misses 105

5.3.2 Ideal False Sharing Detector (I-FSD) 106

5.3.3 Comparison with Dubois' scheme 110

5.3.4 Suitability of I-FSD for On-Line Miss Classification 113

5.3.5 Practical False Sharing Detectors 114

5.3.6 Design Choices for Access Tracking 115

5.3.7 Design Choices for Overlap Detection 118

5.3.8 Non-primary Private Caches 118

5.3.9 Other issues 119

5.3.10 Evaluation 120

5.4 Comprehensive classification 130

5.5 Related Work 132

5.6 Summary 134

VI CONCLUSIONS 135

APPENDIX A ATTRIBUTION OF CACHE MISSES 137

APPENDIX B CACHE MISSES FOR DIFFERENT INPUTS 147

REFERENCES 163

LIST OF TABLES

1 Examples of Correctness bugs and their effects. 2

2 Latency, area, and power overheads, expressed as a fraction of the latency, area, and power of the unmodified 32 KByte L1 cache. 55

3 Meaning of Filter TPT entries 72

4 External Input Tracking. 77

5 Heap Pointer Tracking. 78

6 Latency, area, and energy overheads. 104

7 Program Counters and the corresponding number of false sharing misses reported in Facesim Benchmark. 109

8 Attribution of Replacement Misses. Capacity and conflict misses are shown as percentage of replacement misses. 139

9 Attribution of Replacement Misses. Capacity and conflict misses are shown as percentage of replacement misses. 140

10 Attribution of Replacement Misses.Capacity and conflict misses are shown as percentage of replacement misses. 141

11 Attribution of Replacement Misses. Capacity and conflict misses are shown as percentage of replacement misses. 142

12 Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses. 143

13 Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses. 144

14 Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses. 145

15 Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses. 146

16 Splash-2 Benchmarks with different inputs. 148

17 Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses. 149

18 Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses. 150

19 Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses. 151

20 Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses. 152

LIST OF FIGURES

1 Hardware-Software Ecosystem. 5

2 Tainting an "unsafe" input value and propagating the taint in a Data-Flow Graph of a program segment. A "jump" on a tainted value is detected as unsafe operation. 11

3 The miss on X is a not a false sharing miss — it is only replaced by another miss if X and Y are placed in separate blocks. 18

4 A parallel reduction showing false sharing on an array of counters (one-per-thread). The merging of partial results is omitted for space. 18

5 A parallel histogram computation illustrating false sharing on an indirectly accessed array. Locking of counter_array elements is omitted for brevity. . . 19

6 A red-black Gauss-Seidel-style array update showing false sharing on a finely partitioned array. 20

7 State transition diagram for the HeapData checker. 24

8 Example Instructions with variable footprint (e.g., UEVT0 is used to annotate memory allocation and UEVT1 to annotate memory deallocation) and constant footprint (e.g., store). 28

9 State transition table for our example HeapData checker. Entries with "E" trigger exceptions. 29

10 A dispatcher function F_{AB} calling F_A and F_B to determine the output of checkers A and B respectively. 33

11 Lookup of 2-bit state for data location 0xABCD. 35

12 Caching approaches for MemTracker state in the L1 cache. For L2 and below we always use Shared. 37

13 Processor pipeline with MemTracker support (shaded) for different L1 state caching approaches. 38

14 PSTT setup for the *HeapChunks checker*. 43

15 PSTT setup for the *RetAddr checker*. 44

16 Effect of shared, split, and interleaved caching of state and data in L1 caches. 48

17 Overhead of different checkers in MemTracker, with Split L1 state caching using a 16KByte L1 state cache. 50

18 Performance overhead variation due to different sizes of L1 state cache. . . 52

19 Effect of state changes in software handlers. 52

20 Previous tainting support 65

21 Processor pipeline with FlexiTaint 66

22 Taint propagation in FlexiTaint 66

23 Taint propagation for stores 67

24 Performance overhead of taint propagation with FlexiTaint. 80

25 Use of FlexiTaint optimizations. 81

26 Non-silent taint writes. 82

27 Effect of TL1 line size in Splash-2. 83

28 Effect of limited programmability. 85

29 LRU stack (used by I-CMD) and CMD generational stack. The shading shows how recently a block was accessed. In the LRU stack, there is a total ordering among the blocks in the stack. In the CMD stack, no ordering information is maintained within a generation. 92

30 Shaded areas show the hardware added to implement our Conflict Miss Detector (CMD). A 4-generation CMD is shown, at a time when Gen 2 is currently the youngest one. 94

31 CMD implementation flowchart. 95

32 Percentage of inaccuracy for generational CMD on caches with LRU and Clock-based Replacement policies. 98

33 Correlation coefficient between conflict misses suffered by all static instructions in caches with LRU and Clock Replacement policies. 99

34 Breakdown of replacement misses. Each benchmark has two bars- the left bar shows the classification according to I-CMD and right bar shows the classification according to our practical CMD. 101

35 Percentage of inaccuracy in our practical CMD when thresholds in generations are non-adaptive and adaptive. 102

36 Accuracy of classification for 4, 8 and 16 generations. 103

37 Breakdown of all cache misses for 8 and 64 cores. 105

38 I-FSD Algorithm 107

39 False Sharing Misses in Facesim benchmark. 110

40 Accurate classification requires us to keep access information even for blocks that are no longer in any cache. 111

41 Comparison between I-FSD and Dubois' scheme. Each bar has three portions- bottom portion shows the percentage of times there is agreement between two schemes, middle portion shows percentage times Dubois' scheme misclassifies as false sharing (non-essential) on the producer side and the top portion shows the percentage times Dubois' scheme misclassifies as true sharing because of loss of history regarding *staleness* of values. 112

42 Design Space for coherence miss classification mechanisms. Gradations of AcT state is on the vertical axis and OD state is on the horizontal axis. Hardware costs increase as we move up or towards the right. Design points marked 'x' are examined in our evaluations. 115

43 Accuracy versus cost trade-offs for practical schemes having total OD state and varying amounts of AcT state. For each benchmark, the six bars (from left to right labeled 1 through 6) represent Near-Total AcT state, Partial AcT state with 16k, 4k, 1k, 64 entries and Zero AcT state. 121

44 Correlation coefficient between number of false sharing misses suffered by static instructions in Partial AcT, Zero AcT schemes against Near-total AcT. Samples of static instructions are chosen by Partial AcT schemes. . . 123

45 Percentage of inaccuracy for Zero AcT and different implementations of Near-Total AcT. For each benchmark, there are four bars- Near-Total, Near-Total-2 (does not consider reader information on the writer side), Near-Total-3 (does not consider silent writes and ignores reader information on the writer side) and Zero AcT. 124

46 Accuracy versus cost trade-offs for practical schemes having Near-Total AcT state and varying amounts of OD state. For each benchmark, the four bars (left to right labeled 1 through 4) represent Total OD state, Partial OD state with 1k, 256 entries and Zero OD state. 125

47 Correlation coefficient between number of false sharing misses suffered by static instructions in Partial OD, Zero OD schemes against Total OD. Samples of static instructions are chosen by Partial OD schemes. 127

48 Accuracy versus cost tradeoffs for certain low-cost practical FSD schemes. For each benchmark, the three bars (left to right labeled 1 through 3) represent (Partial (1k) OD, Partial (16k) AcT), (Partial(256) OD, Partial(4k) AcT) and (Zero OD, Zero AcT) points in the design space. 128

49 Correlation coefficient between number of false sharing misses suffered by static instructions in various schemes against (Total OD, Near-Total AcT). Samples of static instructions are chosen by Partial state schemes. 129

50 Cache miss classification for smaller (32 KB) L1 and larger (256 KB) L2 caches. 131

51 Average per-miss pipeline stall cycles incurred by different types of cache misses. 132

SUMMARY

Thesis statement: Efficient architectural support for debugging correctness and performance of programs can be achieved at low cost and minimal performance overheads.

Moore's law has enabled rapid advances in computer hardware. There are billions of transistors inside modern day processors. This increasingly powerful hardware has been exploited by increasingly complex software. As a result, software is prone to a variety of bugs – some of which have even become security exploits. Over the past years, there have been several incidents related to program bugs and their consequences have ranged from monetary losses to costing human lives. *Correctness debugging* ensures that a program does not exhibit any unintended behavior during runtime.

While there can be no second thoughts on whether a program should be correct, ensuring good performance of software is equally important. *Performance debugging* mechanisms are aimed at locating performance bottlenecks and helping to improve program runtime. With multi-core processors becoming rapidly popular in today's market, the need to boost performance is even higher. These multi-core processors have an added value only if the software vendors can take full advantage of the available cores. Hence, scalable program performance is important in these platforms.

Prior works have studied either software-based or hardware-based solutions to address program correctness and performance. Software-based solutions are low-cost and flexible but usually incur very high performance overheads. Hardware-based solutions are efficient with low performance overheads but are usually expensive and inflexible. If we can get

the best of both worlds by leveraging the flexibility and low-cost from software and efficiency from hardware, that would make debugging solutions more attractive for software developers.

In this dissertation, three novel techniques that provide hardware support to facilitate memory debugging, dynamic taint propagation and comprehensive cache miss classification are explored. All of our techniques have two common goals namely low-cost and efficiency. When possible, we incorporate programmability into hardware. This amortizes the cost of hardware by making it usable for multiple purposes. Our contributions define a new direction that renews architects' commitment to build hardware that are more programmer-friendly and help software developers achieve software correctness and performance with ease.

CHAPTER I

INTRODUCTION

1.1 Motivation

Computing has rapidly evolved over the decades as a result of phenomenal growth in hardware, software and communication technologies. Software, in particular, has become increasingly complex which has led to a large number of bugs. There is little value in having a sophisticated program that does not compute its result correctly. Further, global connectivity enabled by the Internet puts computer users at higher risk of being exposed to security attacks especially if their programs have vulnerabilities. Consequently, ensuring program correctness is a significant aspect of the software development cycle.

Table 1 shows a few examples of correctness bugs from the past. These bugs come from a broad spectrum of program errors such as race condition in parallel programs, error in system clock, and integer and buffer overflows. Their consequences are also varied: some are disastrous costing human lives (e.g., Therac-25 [42], Patriot bug [66]) while others are relatively benign (e.g., Twilight Hack [51]). Such incidents deteriorate users' confidence on software and software vendors have to ensure that their products do not cause losses to end-users.

Inadequate support for software testing has been estimated to cost $22 billion annually to the US economy [70]. Hence, tools that ensure correctness of programs are of paramount importance in this era where computers have become accessible to users everywhere. *Correctness debugging* mechanisms attempt to make sure that the program does not exhibit any unintended behavior at runtime.

An *all correct* program *without good performance* is unlikely to lend any commercial success to the software product. *Performance debugging* is an active area of research

Table 1: Examples of Correctness bugs and their effects.

Incident	**Cause and *Effect***
Therac-25 1985-87	Race condition in radiation administering program *cost three human lives.*
Morris Worm 1988	Buffer overflow in gets() function in finger daemon led to *Denial of Service attacks and estimated $100 million loss.*
MIM-104 Patriot Bug 1991	Error in system clock led to clock drift of one-third of a second. Failed to intercept Iraqi scud and *cost 28 human lives.*
Ariane 5 Rocket 1996	Integer Overflow in 64-bit Floating Point to 16-bit integer conversion led to *rocket crash.*
Twilight Hack 2008	Buffer overflow by loading specially crafted save file that *reloads 'The legend of Zelda: Twilight Princess' [51]* *on Wii [52] console*.

that caters to developing tools helping programmers to identify the causes for performance bottlenecks. Performance analysis involves determining whether the program achieves intended performance and is subtly different from performance debugging. Modern processor features like Intel Precise Event Based Sampling (PEBS) [31] and AMD Instruction Based Sampling [21] are incorporated primarily to identify performance bottlenecks in programs.

With the arrival of multi-core processors in the market, the computing landscape has been rapidly redefined for computer architects, software programmers and end-users. Although multiprocessor systems have long been a subject of research for architects, their widespread use has spawned an abundant number of interesting practical problems that await solutions. Multi-core hardware provides the starting point for a paradigm shift in programming newer computing platforms, and programmers are responsible for writing better-running programs and for utilizing multiple cores to help the increased core count have a direct impact on the end-user. In order to harness the full potential of these multi-core architectures, scalability of applications is an important requirement. An end-user who invests in buying an eight-core machine ideally expects his/her application to run eight times faster or the aggregate throughput of the machine to be eight times higher than

a single-core machine. For this reason, application writers who parallelize their programs, are interested in achieving a speedup that equals or approaches the number of cores to satisfy their end-users.

Writing scalable parallel applications can be a daunting task even for seasoned programmers. Until recently, it was mostly the purview of a small group of people who needed to solve computationally expensive problems, had extensive training at parallel programming and had access to very expensive parallel machines. The vast majority of sequential programmers are now suddenly left to face an uphill task of adapting to these new multi-core architectures. Necessary and adequate tools are needed for such programmers to address scalability, adaptability and programmability challenges faced by them on multi-core platforms.

1.2 Architectural Support for Debugging

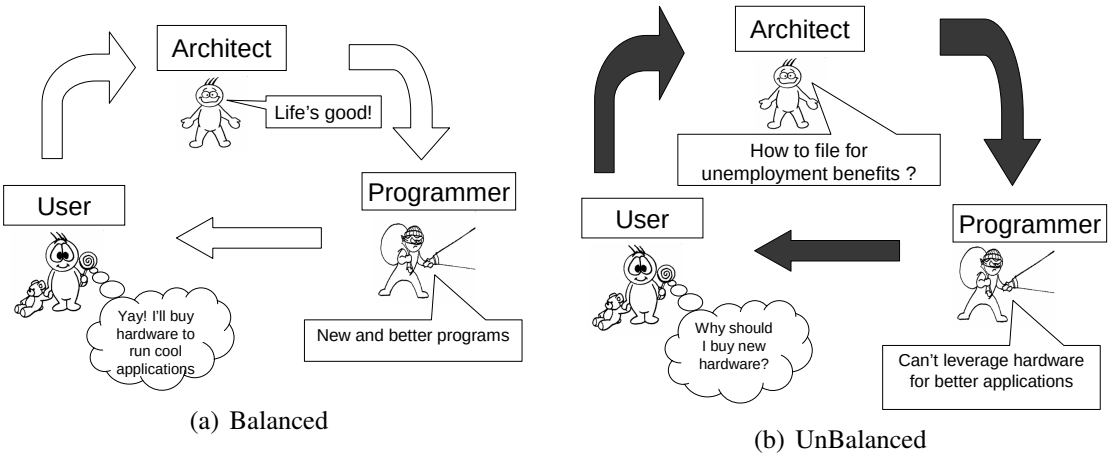
While it is clear that we need tools to address software correctness and performance, trade-offs involving their choice of implementation also needs to be understood. Debugging tools are either software-based or hardware-based. Software-based solutions typically instrument the original program with extra code to monitor program behavior. This provides the advantage of flexibility, that is, the software can be instrumented for any desired debugging task by simply inserting the necessary debugging code. However, as a result of instrumentation, the code size grows and the underlying hardware has to execute additional instrumented code. Hence, these software tools incur huge performance overheads. For example, software-based memory (correctness) debugging tools such as Valgrind [60] and Purify [35] incur very high overheads (up to 30$\times$ slowdown) that prevent their use in production (live) runs. Also, without careful (less intrusive) instrumentation, software-based performance debugging tools (e.g., MTOOL [26]) can potentially alter program behavior and memory access patterns failing to observe a problem that might otherwise occur in real hardware.

Another possibility is to consider simulation-based studies [6, 43, 44]. Simulators model the underlying hardware behavior and programs are tested on such tools. This provides the advantage of simulating program behavior for any given architecture without actually incurring the cost of building actual hardware. Simulations are, however, typically time consuming to evaluate an entire program execution. For example, MemSpy [43] reports up to 57.9× slowdown and SM-prof [6] reports up to 3000× slowdown depending on the type of inputs. Hence, such an approach is feasible only when program inputs are significantly scaled down to keep the simulation time reasonable. This restricts inputs and might considerably decrease programmer's confidence in reaching any suitable conclusions since a correctness bug can go undetected or a performance bug that occurs on realistic inputs might go unnoticed. It is also very difficult to model the actual hardware accurately using simulators.

An alternative to solve the problems posed by software-only solutions is to build custom hardwired logic to detect bugs [17, 20]. These hardwired techniques target specific types of program bugs and can achieve better performance without significant slowdowns. Depending on the nature of the program bug, these solutions are sometimes acceptable for software developers because it helps them avoid expensive errors in programs. However, architects are usually reluctant to invest in specific hardwired schemes because they are usually expensive and require periodic upgrades as new bugs are discovered. Also, hardware manufacturers have to incorporate such hardwired solutions in their future products to maintain backward compatibility for existing programs.

A reasonable trade-off between the two types of solutions would be to take advantage of the low-cost and flexibility offered by software-based solutions while incorporating the efficiency of hardware-based tools. Low cost enables adoption of these tools by a wide range of programmers. This is usually achieved by keeping the amount of changes to hardware as low as possible. Further, programmable hardware amortizes the cost. By providing programmability, we incur the cost of building the new hardware but get the

advantage of using the same hardware for multiple purposes. Efficiency is achieved by not affecting performance-critical parts of the processor. This helps keep the performance impact low and enables usage of such tools after deployment. It is especially useful for software programmers because they can capture bugs after program deployment. A second advantage with efficient debugging tools is that programmers can test their programs with realistic sized inputs.



(a) Balanced (b) UnBalanced

Figure 1: Hardware-Software Ecosystem.

In general, computer architects, software developers and end-users (who depend on programmers for applications) form a dependent ecosystem (Figure 1). Architects design high performance architectures and deliver them to software developers, who in turn, exploit the newly designed hardware features to develop newer and better applications. End-users are enticed by these better applications and invest in buying the hardware that sustains a healthy ecosystem (Figure 1(a)). Once the programmers are unable to leverage the new hardware for better applications, they fail to meet end-users' expectations. This jeopardizes architects' abilities to develop better hardware and destabilizes the balance of the ecosystem (Figure 1(b)).

In essence, end-users are satisfied when the applications are bug-free and performance is good. Software developers are happy when they have appropriate tools to address both

software correctness and performance. Architects have the responsibility to help programmers by providing adequate hardware support to aid debugging. We draw inspiration from here and explore ideas that would form stepping stones in this promising direction.

1.3 Overview

This dissertation contains six chapters.

- Chapter 1 introduces the concepts of correctness and performance debugging, and the reasons why architects should invest effort in building hardware support for debugging.

- Chapter 2 provides necessary background for this dissertation. We introduce specific problems in the domain of correctness debugging, namely, memory debugging and tainting. We then identify an important performance bottleneck in programs, namely, cache misses, and describe why it is important to measure their impact. As we transition to multi-core architectures, the effects of cache misses become even more crucial in making programs scalable. Cache misses caused by false sharing are a major scalability limiter, and we provide a *programmer-centric* definition along with real-world examples of false sharing misses.

- Chapter 3 describes MemTracker, a programmable hardware mechanism for detecting memory-related bugs. MemTracker provides a high degree of flexibility to users for implementing several different memory checkers and even combining them at ease. MemTracker's low ($<$5%) execution time overheads enable its use in live runs.

- Chapter 4 describes FlexiTaint, a programmable accelerator for taint propagation in hardware. Tainting is a popular Dynamic Information Flow Tracking mechanism to track certain values during runtime. Its applications range from tagging untrusted values and raising alarms when used in spurious ways [17], to tracking heap-based pointers for accelerating memory leak detection. Our evaluation results show that

FlexiTaint offers flexibility to track completely different taint propagation policies simultaneously at low (<5%) execution time overheads.

- Chapter 5 describes CacheDoc, a comprehensive cache miss classification study. Cache misses are usually categorized as cold misses (when cache blocks are accessed for the first time), replacement misses (when cache blocks replace other cache blocks because of insufficient capacity or due to conflicts arising from mapping several cache blocks to the same set) and coherence misses (when cache blocks are shared between cores and are involved in coherence actions). We explore a series of design points for classifying replacement and coherence misses and study accuracy versus cost trade-offs involved in these designs.

- Chapter 6 presents conclusions of this work.

1.4 Scope of this Dissertation

The main goal of this work is to motivate the reader with the need for tools that can help solve some of the widely known, and some of the emerging issues in software correctness and performance. We describe how our proposed solutions can be integrated into modern processors by outlining the necessary hardware modifications to incorporate them. We evaluate our ideas through simulations that model the modified processor platforms and we study the results. Our work shows promising trends in achieving low-cost and efficient solutions for some of the important problems in software debugging. We envision this dissertation as a beginning to a long road which will have a plethora of efficient and inexpensive solutions for debugging programs.

CHAPTER II

BACKGROUND

In this chapter, we introduce two well-known problems in correctness debugging and one important problem in performance debugging. Specifically, we first describe the need for memory debugging, the merits and demerits of existing techniques and why we need programmable hardware support to detect memory bugs. Second, we describe how tainting can be used as a dynamic data flow tracking mechanism and discuss some of the previous works that perform taint propagation in software and in hardware. Third, we identify cache misses as a performance bottleneck and describe the need to diagnose and classify these cache misses. With the increasing significance of multi-core processors, cache misses caused by false sharing are especially detrimental to the scalability of parallel programs on these platforms. We present a "programmer-centric" definition of false sharing misses and provide several real-world examples where scalability of applications is affected by these misses.

2.1 Memory Debugging

Increasing software complexity enabled by rapid improvements in hardware technology and architecture, has resulted in a wide range of programming bugs. One particularly important and broad class of programming errors is erroneous use or management of memory (memory bugs). This class of errors includes pointer arithmetic errors, use of dangling pointers, reads from uninitialized locations, out-of-bounds accesses (e.g., buffer overflows), memory leaks, etc. Many software tools have been developed to detect some of these errors. For example, Purify [35] and Valgrind [60] detect memory leaks, accesses to unallocated memory, reads from uninitialized memory, double frees, freeing of statically allocated

memory, and some dangling pointer and out-of-bounds accesses. Some memory-related errors, such as buffer overflows, are also a well-known source of security vulnerabilities. As a result, security tools often focus on detection of such errors or specific error manifestation scenarios. For example, StackGuard [16] detects buffer overflows that attempt to modify a return address on the stack.

In order to monitor memory-related bugs, a software detection tool (checker) must *intercept* memory accesses (loads and/or stores) and perform the following set of actions:

- *State lookup*: For each access, the checker must find the state of the memory location, e.g., determine whether the location is allocated, initialized, stores a return address, etc.

- *State check*: Once the state is obtained, the checker must verify if the access is allowed for a memory location with such state, e.g., determine whether a load reads from an initialized memory location.

- *State update*: For certain accesses, the checker could potentially update the state of the memory location, e.g., a write access changes an uninitialized memory location into an initialized one.

Since memory read and write instructions are executed frequently, the overhead of intercepting and checking them is very high. Slowdowns of $2\times$ to $30\times$ have been reported for some popular software tools like Valgrind [73].

Architectural support has been proposed to reduce performance overheads for detecting some memory-related problems [62, 77, 81, 82, 83]. Many of these schemes allow loads and stores to be intercepted in hardware without inserting instrumentation instructions around them. After intercepting an access, a checker still needs to perform a state check and possibly a state update. Previously proposed schemes disagree on whether state checks and updates should also be performed in hardware, because that decision determines the trade-off between performance and the ability to support different checkers. One

approach is to hardwire the meaning of each state for a specific checker [81], which allows one specific checker or a family of checkers to be implemented very efficiently. Another approach is to perform interception in hardware and dynamically insert software handlers for state checks and updates [15]. Finally, a number of existing approaches express state as access permissions or monitored regions. Such state can quickly be checked in hardware to determine whether the access can proceed without any additional checker activity. If a state update (or additional checking) is needed, a software handler is invoked [62, 77, 82, 83]. Overall, existing architecture support is either i) hardwired for a particular checker, or ii) requires software intervention for *every state update*. This can lead to significant performance degradation for checkers with frequent state updates. In our approach named MemTracker (described in Chapter 3), we overcome the above-mentioned problems by providing a programmable substrate to implement many different memory checkers along with the hardware performing most of the memory checks.

2.2 Tainting

While memory debugging (described in Section 2.1) specifically targets program bugs associated with memory-related events such as loads, stores, allocation and deallocation of memory buffers, access permissions, etc., memory checkers cannot fully guarantee software correctness. A growing concern in the software community is the increasing number of security threats posed by program bugs that can be used as exploits. To help deal with these problems, a variety of runtime checking and tracking approaches have been proposed. A number of these proposals have adopted dynamic taint propagation, or *tainting*. Typically, a tainting scheme associates a *taint* with every data value. Figure 2 shows an example of taint propagation. The taint is usually a one-bit field that tags the value as safe (untainted) or unsafe (tainted). Data from trusted sources starts out as untainted, whereas data from an untrusted source (e.g. network) starts out as tainted. Taints are then propagated as values are copied or used in computation. To detect potential attacks, a tainting scheme looks for

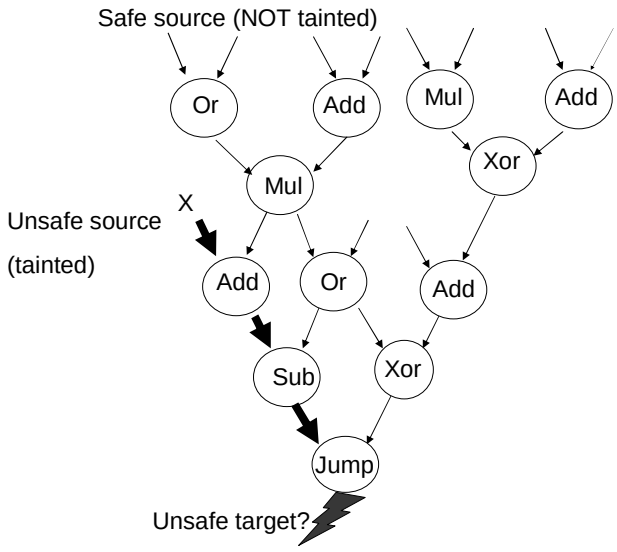


Figure 2: Tainting an "unsafe" input value and propagating the taint in a Data-Flow Graph of a program segment. A "jump" on a tainted value is detected as unsafe operation.

unsafe uses of tainted values. For example, using a tainted value as a jump target address is considered unsafe because such a jump may allow the attacker to hijack the control flow of the application.

Tainting, in general, can be used as a generic Dynamic Information Flow Tracking (DIFT) technique to tag data values of interest and track their flow through program execution. Taint propagation is performed either in software or hardware. Several software-based taint propagation schemes have been proposed as comprehensive solutions against specific types of security attacks [32, 50, 53, 61, 79]. However, software-only schemes have large performance overheads and have significant problems with self-modifying code, JIT compilation, and multi-threaded applications. Multi-threaded applications are especially difficult due to potential race conditions between data and the corresponding taint value updates. Hardware-assisted schemes try to avoid these drawbacks [39, 18, 10, 68, 64].

Many hardware tainting schemes suffer from two problems that limit their practicality. First, their implementations require non-standard commodity components and a redesign of most of the core's datapath. Since taint bits are added to every value in memory and in the processor, wider memory, registers, buses and bypasses are needed. The second

problem is limited flexibility in specifying taint propagation rules, and in the number of taint bits associated with a value. Most schemes [50, 79] provide single-bit taints with little or no flexibility in how that taint is propagated. Those that allow multi-bit taints do so by significantly increasing the implementation cost, as memory modules, buses, and the processor core must be designed to accommodate the largest supported taint [18]. Similarly, schemes that provide some flexibility in specifying taint propagation rules are still limited to simple rules that mainly target only one particular use of tainting, e.g., tainting of inputs to detect attacks. In our approach named FlexiTaint (described in Chapter 4), we overcome the above-mentioned disadvantages by i) using off-the shelf memory modules and through minimal modifications to the processor core and ii) providing programmable hardware to implement several different taint propagation policies.

2.3 Performance Debugging

While ensuring program correctness through memory debugging and tainting is very important, maximizing the performance benefits offered by current processors is also critical to a successful software product. Some of the performance bottlenecks are related to the application itself, e.g., the number of instructions that can be executed simultaneously at any given time, popularly referred to as Instruction Level Parallelism. Many other bottlenecks result from the application's interaction with the underlying hardware. Such bottlenecks include excessive cache misses, mispredicted branch instructions, etc.

With the growing popularity of modern day multi-core architectures, tapping their potential requires efficient use of the multiple cores on the chip. Applications must be written so that their performances scale linearly, or as close to linearly as possible, with the number of available cores. This is often extremely challenging, even when the underlying application is amenable to parallelization. Parallel applications have a number of possible performance problems that either do not exist in serial applications or that can be aggravated by the sharing of resources.

Processor vendors understand that making processors with increased raw performance is not sufficient: customers must see a real performance difference. This requires that software vendors write their applications to take advantage of the increased raw performance. This is not a trivial task, and therefore, most processor vendors currently support a number of performance counters to aid this task [36]. Performance counters aid programmers in finding and eliminating performance bottlenecks in their code (i.e., performance debugging). Since it is much harder to realize the performance potential of multi-core and many-core processors, processor vendors should be willing to invest some additional resources to aid in performance debugging issues introduced with multi-core and many-core processors. Ideally, these resources would include performance counters for additional multi-core and many-core events, and hardware to detect those events.

A natural question to ask is whether hardware support brings significant value to performance debugging given the wide variety of software-only performance debugging tools available. For at least three aspects, the answer is "absolutely." Tools that interface with the performance counters are extremely popular because they i) can collect information on an enormous variety of performance bottlenecks and present it in a usable way (i.e., pinpoint problems in the source code), ii) are very fast, and iii) are very accurate. In contrast, software-only tools i) typically only measure a small set of events since they must trade off execution time overhead and the amount of information collected, ii) are generally slow even when only measuring a small number of events (e.g., SM-prof [6] reports up to 3000$\times$ slowdown depending on the amount of shared data references), and iii) are not reliably accurate because in many cases they cannot model hardware perfectly and/or perturb the program execution by interleaving software instrumentation into that execution. Having a tool that is fast, accurate, and can measure a number of events of interest is extremely valuable.

Memory accesses that miss in caches frequently consume additional latency and degrade program performance. While cache misses certainly affect the performance in single

core machines, their effects are even more pronounced in multi-core processors. A variety of memory system behaviors that result in poor application scaling manifest themselves as additional cache misses. Therefore, determining the source of the additional misses can be of great help to a programmer. The "source" of a cache miss includes both the underlying reason why the hardware did not have the line in the cache as well as the place in the code that triggered the miss.

Hardware may incur additional cache misses for parallelized applications for a number of reasons. The most common ones for shared memory systems are: i) Reduced temporal and spatial locality from input data partitioning, ii) inter-thread communication, via both true sharing (i.e., intentional communication) and false sharing (i.e., unintentional communication), and iii) destructive sharing (i.e., additional conflict misses from cache sharing).

False sharing and destructive sharing, in particular, are issues that often surprise programmers; without sufficient knowledge of the underlying hardware, the presence of these misses is mysterious and even frustrating. They can also have a devastating impact on parallel scalability (Section 2.4). For example, programmers often use an array of counters or accumulators (one per-thread) when parallelizing reductions. Without appropriate padding, if these structures are frequently accessed, the application will incur a huge increase in cache misses due to false sharing. Another frequent programmer behavior is partitioning a data structure into power-of-two-sized chunks. If the chunks are aligned in a shared cache (i.e., map to the same sets), and the cache's associativity is not sufficient, the application will incur a huge increase in cache misses due to destructive sharing [29]. This also frequently occurs if threads' stacks are power-of-two aligned.

One way to help distinguish between the above three causes of cache misses is to classify misses using the common categories: i) compulsory, ii) capacity, iii) conflict, and iv) coherence [30]. A cold miss occurs when the requested block has never been in the cache before. Capacity and conflict misses are collectively called replacement misses. These misses occur when a block is re-accessed after it has already been fetched into the cache

and then evicted to accommodate another block. Capacity misses are replacement misses due to limited cache size. Conflict misses are caused by limited cache associativity; these would be hits in a fully associative cache of the same size. Finally, coherence misses are caused by sharing of data between cores. These misses occur when a block is invalidated or downgraded in a core's cache, and then re-accessed by that core. Unlike the other three classes of misses (compulsory, capacity, and conflict), a coherence miss finds the block in the cache, but the block is in a non-usable state: a read finds the block in an invalid state, or a write finds the block in invalid or shared states.

Classifying cache misses into these categories can help identify the root cause of the misses. For example, a large jump in conflict misses in a shared cache between a single-threaded run and a multi-threaded run likely indicates destructive sharing. It is also helpful to additionally classify coherence misses as being caused by true or false sharing since these have fundamentally different sources, and are addressed by the programmer in different ways. Cache miss classification helps programmers to use techniques, such as array grouping to reduce communication delays [63] or cache conscious data placement [8, 11], to minimize the effect of such misses. Although our aim is to primarily help programmers, we derive other potential benefits from identifying various types of cache misses. For example, victim caches can choose to store blocks that frequently suffer conflict misses [13]. Prefetchers can improve performance by bringing in blocks that suffer a large number of capacity misses.

We note that performance of applications can be improved only if programmers, compilers, and/or dynamic optimizers carefully apply appropriate fixes to reduce the number of cache misses in the program. Different types of cache misses require different types of fixes, and knowing the dominant type of cache misses is helpful when deciding on an approach to fixing them. For example, conflict misses and false sharing misses are frequently addressed through padding the memory, while true sharing misses might require fundamental redesign of the algorithm. Although there is no doubt that the effort required

differs depending on the type of miss, not all misses of a particular type are amenable to the same fixes. For example, fixing false sharing misses by padding a data array may introduce additional cache misses due to conflict with other cache blocks. Hence, depending on the type of miss and its sorrounding access patterns, potential performance improvement is likely to vary across different cases. Nevertheless, identifying the sources and causes (type) of these cache misses helps programmers to understand program behavior and to decide which techniques to use for improving program performance.

There have been several proposals for cache miss classification off-line or in architectural simulators [30, 44, 71]. A pure software-based scheme to dynamically classify cache misses can degrade the application performance by several orders of magnitude [6]. This is unacceptable not only because it is slow but also because it can significantly alter the execution path of a parallel program, minimizing the utility of the generated cache miss classification. Collins et al. [13] have proposed a hardware scheme that identifies conflict misses in a cost-effective manner. However, their definition of a conflict miss differs from the classical definition and is primarily directed towards helping prefetchers and victim caches. To our knowledge, our scheme named CacheDoc (described in Chapter 5) is the first to provide a comprehensive cache miss classification mechanism that can be performed on-the-fly to drive performance counters and hardware-assisted profiling.

2.4 False Sharing and its Implications on Scalability

In Section 2.3, we analyzed the need to classify cache misses into compulsory, replacement and coherence misses. Coherence misses, in particular, are becoming increasingly relevant for multi-core processors and needs to be better understood by programmers who are much more familiar with compulsory and replacement misses that occur in single core processors. In this section, we provide a programmer-centric definition of false sharing and true sharing misses and then discuss several real-world examples where false sharing misses occur and how removing these misses improves scalability.

2.4.1 Definition

In cache coherent CMPs, data sharing between threads primarily manifests itself as coherence misses which can further be classified as true sharing and false sharing misses.

True sharing misses are a consequence of actual data sharing among cores and are intuitive to most programmers, e.g., a consumer (reader) of the data must suffer a cache miss to obtain the updated version of the data from the producer (writer). The scalability issues stemming from true sharing misses can typically be addressed only by changing the algorithm, distribution of the work among cores, or synchronization and timing.

In contrast, false sharing misses are an artifact of data placement and a cache block holding multiple data items. Scalability issues arising from false sharing are often relatively easy to alleviate by changing alignment or adding padding between affected data items. A false sharing miss occurs if a block contains two data items (X and Y), core A accesses item X, core B uses item Y and invalidates the block from A's cache, and then core A accesses item X again. The resulting cache miss is a false sharing miss because that access would be a cache hit if X and Y were in different cache blocks. Note that, if after the coherence miss on an item X, core A accesses item Y before the block is evicted or invalidated again, the miss is in fact a true sharing miss. As shown in Figure 3, in this situation, the cache miss is not avoided by placing X and Y in separate blocks, and hence, the miss on X is not a false sharing miss. This definition of false sharing is based on Dubois et al. [23], and it differs from earlier definitions [24, 71] in that it attempts to classify coherence misses according to whether they are "essential" (true sharing) or "non-essential" (false sharing). We adopt this definition as a baseline because it more accurately captures whether or not the miss can be eliminated (not just traded for another miss) by separating data items into different blocks which, in the end, is what really matters for the programmer's performance debugging efforts.

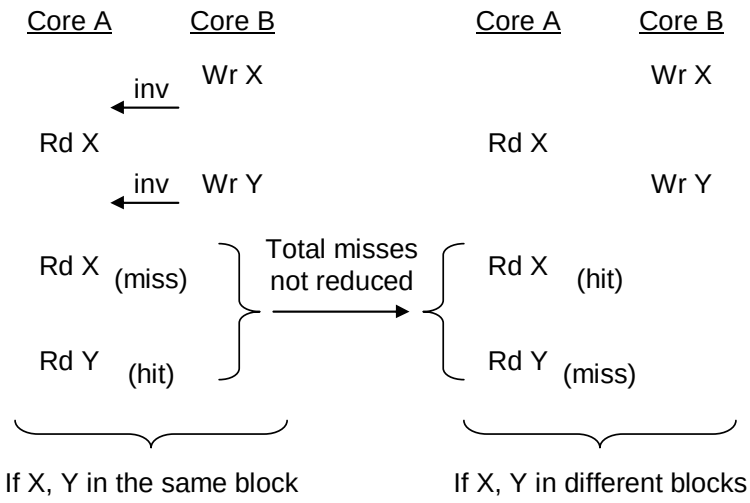


Figure 3: The miss on X is a not a false sharing miss — it is only replaced by another miss if X and Y are placed in separate blocks.

```
// Each thread processes its own
// sequence of input elements
int partial_result[NUM_THREADS];
// This is the work done in parallel
...
int input_element=...;
partial_result[thread_id] += input_element;
...
```

Figure 4: A parallel reduction showing false sharing on an array of counters (one-per-thread). The merging of partial results is omitted for space.

2.4.2 Real-World examples

We provide examples, taken from code written by experienced programmers, to illustrate some common situations where false sharing occurs and its sometimes devastating impact on parallel scalability.

Our first example involves an array of private counters or accumulators (one per-thread), which is often used when parallelizing reductions as shown in Figure 4. There is no true sharing in this code because each thread is reading and writing a unique array element. False sharing occurs when two threads' counters lie in the same cache block. This kind

```
// Count occurrences of values in parallel
// Each thread processes its own range of
// input elements, updating the shared
// occurrence count for each element's value

#define MAXIMUM 255
int counter_array[MAXIMUM+1];
// This is the work done in parallel
...
int input_element=...;
  counter_array[input_element]++;
...
```

Figure 5: A parallel histogram computation illustrating false sharing on an indirectly accessed array. Locking of counter_array elements is omitted for brevity.

of code is common in web search, fluid simulation, and human body tracking applications [22]. A common fix is to add padding around each counter. We provide a real-world example from a loop in the *One_Newton_Step_Toward_Steady_State_CG_Helper_II()* function from the facesim benchmark in the PARSEC-1.0 suite. The benchmark authors spent multiple days identifying false sharing from this loop as the primary source of performance problems [22]. We ran the facesim benchmark with the native input on an 8-core Intel Xeon machine and observed that without padding, false sharing limits the benchmark's parallel scaling to 4× (Section 5.3.2). After adding padding, the benchmark achieves linear scaling (8×). A profiling tool that automatically identifies and reports false sharing misses would have greatly helped the programmer.

Our second example involves an indirectly accessed data array, as shown in Figure 5 and often occurs in histogram computation, used in image processing applications and in some implementations of radix sort [22]. The pattern of indirections is input-dependent, so the programmer and compiler cannot predetermine how many accesses will occur to each element, and which threads will perform them. This example involves both true and false sharing. True sharing occurs when two threads update the same element. False sharing occurs when two threads access two different elements in the same cache block, which occurs much more frequently. For example, with 64-byte blocks and 4-byte elements, a

```
// The task of each thread is to update one row of the grid
...
for (i = (iteration%2);i < width;i += 2) {
  float val = grid[task_id  ][i  ] + grid[task_id  ][i-1] +
              grid[task_id  ][i+1] + grid[task_id-1][i  ] +
              grid[task_id+1][i  ];
  grid[task_id][i] = val / 5.0;
}
...
```

Figure 6: A red-black Gauss-Seidel-style array update showing false sharing on a finely partitioned array.

block contains 16 elements; with a completely random access pattern, false sharing would occur 15 times more likely than true sharing. A common fix is to either add padding around each element or use privatization (use a separate array for each thread and then merge partial results at the end). Without privatization, a histogram benchmark from a real-world image processing application achieves only a 2× parallel speedup when run on a 16-core Intel Xeon [36] machine. With privatization, the benchmark achieves near-linear scalability.

Our final example of false sharing involves finely partitioned arrays, such as one might find in red-black Gauss-Seidel computations as shown in Figure 6. In many applications, a data array is partitioned such that each partition will only be written to by a single thread. However, when updating elements around the boundary of a partition, a thread sometimes needs to read data across the boundary (i.e., from another partition). In our example, synchronization on each element is avoided by treating the data as a checkerboard, with alternating red and black elements. Even-numbered passes update red cells, and odd-numbered passes update black cells. An update involves reading the Manhattan-adjacent neighbors, which are guaranteed to be a different color than the cell being updated. Thus, even if those neighbors belong to another partition, they will not be updated during the current pass.

Both true and false sharing occurs in this example. The first time a thread accesses a cache block across a partition boundary, it is reading data written by another thread during

the previous pass. Thus, it incurs a true sharing miss. However, if that other thread is actively updating elements in the same cache block, this will trigger additional misses that are all due to false sharing. This situation is most likely when partitions are small; for example, if a parallel task is to update one row, and the tasks are distributed to threads round-robin. This kind of computation is common in scientific codes (i.e., applications that involve solving systems of differential equations) [22]. We ran an early real-world implementation[1] of red-black Gauss-Seidel with the above task distribution on an 8-core Intel Xeon processor. Parallel scaling is limited to 3x. Padding around each cell can eliminate false sharing, but with significant loss in spacial locality. A better solution is to group multiple rows together to be processed by the same thread, or to use a two separate arrays, one for red and one for black cells. In our case, using separate arrays and grouping rows improved the scaling to almost 6x.

2.4.3 Summary

This chapter introduced two important classes of problems in correctness debugging, namely, memory debugging and taint propagation. Cache misses were identified as a key performance bottleneck, and real-world examples were shown where false sharing misses can result in poor application scalability in multi-core architectures. We also discussed several related works, and how that work affects correctness and performance debugging.

[1]We gratefully acknowledge researchers at Applications Research Lab, Intel Corporation for providing us with an early version of Gauss-Seidel benchmark.

CHAPTER III

MEMTRACKER: MEMORY ACCESS MONITOR AND DEBUGGER

3.1 Motivation

Designing architecture support for tracking memory access behavior involves trade-offs between two choices: to sacrifice performance by designing hardware support that is not checker specific, or to sacrifice generality by hard-wiring specific checks for performance. Unfortunately, both alternatives are unappealing: users are reluctant to enable memory access tracking mechanisms that have significant performance overheads, while architecture designers are unwilling to implement hardware that is checker-specific. To overcome this problem, we propose a hardware mechanism, which we call MemTracker, that can *perform interception, state checks, and state updates in hardware*, while still remaining generic and not hard-wired for any particular checker.

MemTracker is essentially a programmable state machine. It associates each memory word with a *state* and treats each memory action as an *event*. States of data memory locations in the application's virtual address space are stored as an array in a separate and protected region of the application's virtual address space. Each event results in looking up the state of the target memory location, checking if the event is allowed given the current state of the location, and possibly raising an exception or changing the state of the location. To control state checking and updates, MemTracker uses a *Programmable State Transition Table* (PSTT).

The rest of this chapter is organized as follows: Section 3.2 presents an overview of our MemTracker mechanism, Section 3.3 presents some hardware implementation details, Section 3.4 presents the setup for our experimental evaluation, Section 3.5 presents our

experimental results, Section 3.6 discusses related work, and Section 3.7 summarizes our findings.

3.2 Overview

To provide context and a motivating example for our discussion of MemTracker, we first describe an example checker. We then describe our MemTracker mechanism and how it can be used to implement the example checker.

3.2.1 HeapData: an Example Memory Access Checker

Many typical programming mistakes, such as use of dangling pointers or neglecting to initialize a variable, are manifested through accesses to unallocated memory locations or to loads from uninitialized locations. Detection of such accesses is one of the key benefits of tools such as Purify [35] and Valgrind [60], and has also been used to evaluate hardware support for runtime checking of memory accesses in HeapMon [62]. To help explain our new MemTracker support, we will use *HeapData*, an example checker that is functionally similar to the checker used in HeapMon. This checker tracks the allocation and initialization status of each word in the heap region using three states: *Unallocated*, *Uninitialized*, and *Initialized*. The state transitions for this checker are shown in Figure 7. All words in the heap area start in the *Unallocated* state. When a block of *Unallocated* memory words is allocated (e.g. through `malloc()`), the state of each word changes to *Uninitialized*. The first write (e.g. using a store instruction) to an *Uninitialized* word changes its state to *Initialized*. The word then remains in the *Initialized* state until it is deallocated (e.g. through `free()`), at which time it changes back to the *Unallocated* state.

Only an *Initialized* word can be read, and writes are allowed only to *Uninitialized* or *Initialized* words. Memory allocation is valid only if all allocated words are in the *Unallocated* state, and deallocation is valid only if all deallocated words are either *Uninitialized* or *Initialized*. All other reads, writes, allocations, and deallocations are treated as errors and should result in invoking a software error handler.

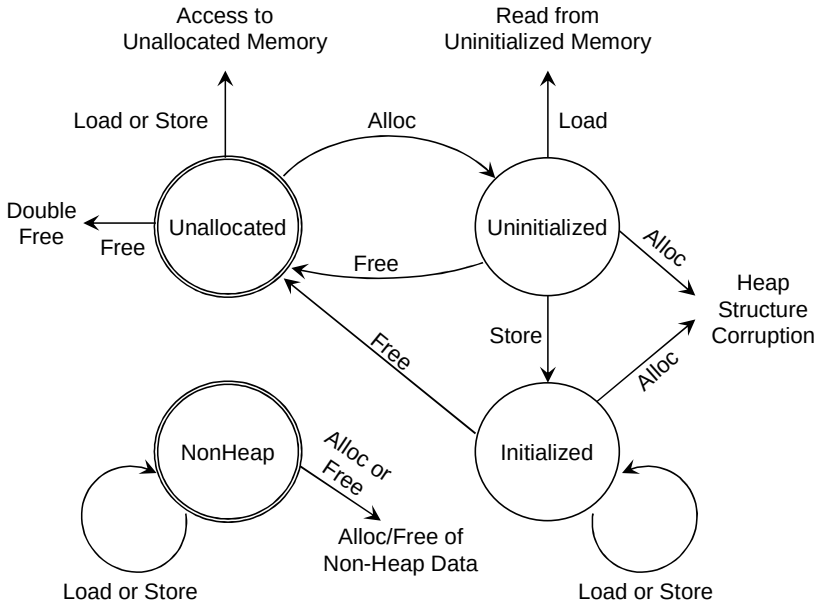


Figure 7: State transition diagram for the HeapData checker.

In addition to these three states from HeapMon [62], our *HeapData* checker has a *Non-Heap* state which is used for all data outside the heap region. This new state allows us to treat all memory accesses in the same way, without using an address-range filter [62] to identify only heap accesses. The *NonHeap* state permits loads and stores, but prohibits heap allocations and deallocations. The Operating system can identify which pages are for the heap because the heap segment is typically grown through the `brk` system call. Therefore, it can initialize heap memory locations to *Unalloc* and non-heap locations to *NonHeap*.

Overall, our example *HeapData* checker reacts to four different kinds of events (loads, stores, allocations, and deallocations), and keeps each word in the application's data memory in one of four different states.

We note that HeapData and other checkers in our experiments are used only to illustrate how MemTracker can be used and to evaluate MemTracker's performance. We do not consider the checkers themselves as our contribution, but rather as familiar and useful examples that help us demonstrate and evaluate our real contribution – the MemTracker

mechanism.

3.2.2 Functionality

For each word of data, MemTracker keeps an entry that consists of a few bits of state. These state entries are kept as a packed array in main memory, where consecutive state entries correspond to consecutive words of data. This packed array of state entries is stored in the application's virtual address space, but in a separately mapped (e.g. via `mmap`) region. Whenever a new physical page is mapped for data, the operating system has to determine whether a physical page exists for the corresponding state addresses. If not, a new physical page is mapped for state information. This operation needs to performed for statically mapped pages at the time of loading the program as well as for dynamically mapped pages in the heap section of the program where pages are mapped on demand. In order to prevent accidental or malicious overwrites to pages containing state information, we provide an extra "state permission bit" apart from the normal Read, Write, Execute permission bits. Pages containing "state" information have state permission bit set and the other permission bits are turned off. This permits MemTracker to access the state pages safely and restricts normal load and store instructions from accessing the state information. This approach avoids custom storage for state and benefits from existing address translation and virtual memory mechanisms. Also, we can use existing structures like Translation Lookaside Buffer (TLB) to perform virtual address translation for state addresses. We provide a "state" permission to TLB in addition to the read, write, execute permission bits. If regular load/store accesses a "state" page directly, an exception is raised. For memory pages that are shared between multiple processes, there are two possibilities - i) all processes could share state and, hence the OS can allocate the same physical page, or ii) each process could choose to maintain state for the shared page separately and, hence the OS can allocate a separate physical page for each process.

When an event (e.g., a load) targets a memory location, the state entry for that location

can be found using simple logic (see Section 3.3.2). Our MemTracker mechanism reads the current state of the memory location and uses it, together with the type of the event, to find the corresponding transition entry in the *programmable state transition table* (PSTT). Each entry in PSTT specifies the new state for the memory location, and also indicates whether to trigger an exception. Entries in the PSTT can be modified either through invoking Operating System or trusted software, allowing MemTracker to implement different checkers.

3.2.3 Events

MemTracker treats existing load and store instructions as events that trigger state lookups, checks, and updates. Other events, such as memory allocation and deallocation in HeapData, should also be able to affect MemTracker state. However, these high-level events are difficult to identify at the level of the hardware and differ from checker to checker. To support these events effectively, we extend the ISA with a small number of *user event* instructions. User event instructions can be used in the code to "annotate" high-level activity in the application and library code. The sole purpose of these instructions is to be MemTracker events. These instructions only result in MemTracker state lookups, checks, and updates, and do not actually access the data associated with that state. The number of different user level instructions supported in the ISA is a trade-off between PSTT size (which grows in proportion to the number of events) and sophistication of checkers that can be implemented (more user event instructions allow more sophisticated checkers). Note that, if binary compatibility with existing systems that do not implement MemTracker is a design requirement, existing no-op or unused opcodes in the ISA should be carefully selected to implement user events. Similarly, for new systems without MemTracker support, user event opcodes should be treated as no-ops. In this work, we model MemTracker with 32 user event instructions, which is considerably more than what is actually needed for the checkers used in our experimental evaluation.

In terms of the number of affected memory locations, MemTracker must deal with two kinds of events: constant-footprint and variable-footprint events. An example of a constant-footprint event is a load from memory, where the number of accessed bytes is determined by the instruction's opcode. The handling of these events is straightforward: the state for the affected word or set of words (e.g. for a double-word load) is looked up, the transition table is accessed, and the state is updated if there is a change. An example of a variable-footprint event is memory allocation, in which a variable and potentially large number of locations can be affected. We note that variable-footprint events can be dynamically replaced by loops of constant-footprint events, either through binary rewriting or in the processor itself in a way similar to converting x86 string instructions into μops [31].

In this work, we use a RISC processor without variable-footprint (e.g. string) load and store instructions, but we provide support for variable-footprint user events to simplify implementation of checkers. Instructions for variable-footprint user events have two source register operands, one for the base address and the other for the size of the target memory address range. This implementation allows simpler checker implementations, and avoids code size increase and software overheads of looping. However, in our simulation, each variable-footprint event is treated as a series of constant-footprint events with each accessing one word at a time. Hence, the overheads due to variable-footprint events are fully accounted for. In fact, these overheads are likely overestimated because more efficient implementations (e.g. handling a double word or an entire cache block at a time) of variable-footprint events are possible. In our MemTracker implementation, 16 of our 32 user event instructions are variable-footprint, and the rest are word-sized constant-footprint events (sub-word events are discussed in Section 3.2.5). Note that we only need two variable-footprint and five constant-footprint user events to simultaneously support all checkers used in our evaluation. The remaining user events are there to provide support for more sophisticated checkers in the future.

3.2.4 Setup for the HeapData Checker

To implement the *HeapData* checker from Section 3.2.1, we use two variable-footprint user event instructions, UEVT0 for allocations and UEVT1 for deallocations. We instrument the user-level memory allocation library to execute UEVT0 at the end of the allocation function, and UEVT1 at the start of the deallocation function. Figure 8 gives a simple illustration for how variable footprint instructions like UEVT0 and UEVT1 are used by our example heap checker.

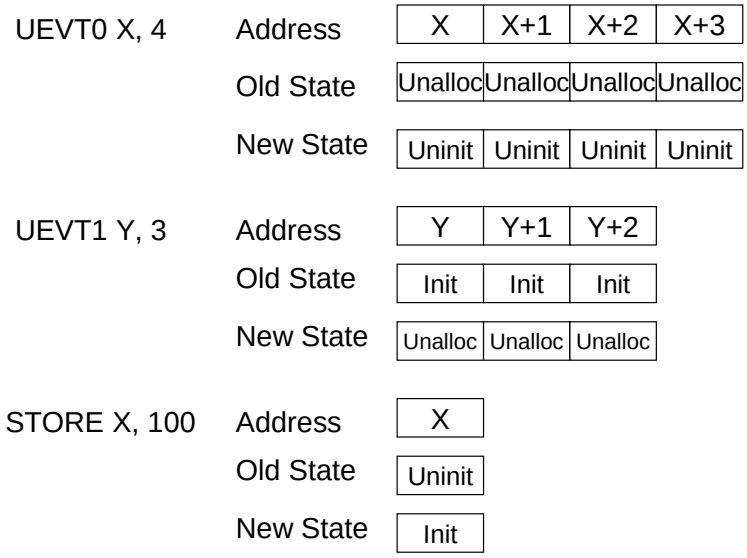


Figure 8: Example Instructions with variable footprint (e.g., UEVT0 is used to annotate memory allocation and UEVT1 to annotate memory deallocation) and constant footprint (e.g., store).

As explained in Section 3.2.3, a variable-footprint event instruction takes the starting address and the number of affected memory words, and it performs the state lookup, check, and update for each of those memory locations. For example, UEVT0 is used as an `allocation of memory` event in the HeapData checker, and in Figure 8 we show how it changes the state of 4 words of memory at address X from *Unallocated* to *Uninitialized*. Similarly UEVT1 is used to denote a `freeing of memory` event, and is shown

in Figure 8 for 3 words at address Y. As an example of an instruction with constant memory footprint, we also show a `store` instruction to memory location X, which changes its state from *Uninitialized* to *Initialized*. Figure 9 shows the PSTT configuration for the heap checker, which is a tabular equivalent of states and transitions described in Section 3.2.1, except for the sub-word LD/ST events, which we discuss in Section 3.2.5.

Event / State	UEVT0 (Alloc)	UEVT1 (Free)	LD	ST	Sub-word LD	Sub-word ST
0 (NonHeap)	0 E	0 E	0	0	0	0
1 (Unalloc)	2	1 E	1 E	1 E	1 E	1 E
2 (Uninit)	2 E	1	2 E	3	2 E	**2 or 3**
3 (Init)	3 E	1	3	3	3	3

Figure 9: State transition table for our example HeapData checker. Entries with "E" trigger exceptions.

It should be noted that the need to modify the memory allocation library is not specific to MemTracker – all tools or mechanisms that track allocation status of memory locations require some instrumentation of the memory management library to capture allocation and deallocation activity. Compared to software-only tools that perform such checks, MemTracker-based implementation has the advantage of eliminating the instrumentation of load/store memory accesses and the associated performance overhead. Even for applications that frequently perform memory allocations and deallocations, the number of loads/stores still easily exceeds the number of allocations and deallocations[1], and hence even such applications benefit considerably from MemTracker.

3.2.5 Dealing with Sub-Word Accesses

MemTracker keeps state only for entire words, so sub-word memory accesses represent a problem. For example, consider the shaded transition entry in Figure 9, which corresponds

[1] In fact, several load/store instructions are executed during each heap allocation and deallocation

to a sub-word store to an *Uninitialized* word. Since the access actually initializes only part (e.g. the first byte) of the word, we could leave the word in state 2 (*Uninitialized*). However, a read from the same part of the word (first byte) is then falsely detected as a read from uninitialized memory. Conversely, if we change the state of the word to 3 (*Initialized*), a read from another part of the word (e.g. the last byte) is not detected as a problem, although it is in fact reading an uninitialized memory location.

In this trade-off between detecting problems and avoiding false positives, the right choice depends on the circumstances and the checker. To allow flexibility in implementing checkers, sub-word load/stores are treated as separate event types in the PSTT. This allows us to achieve the desired behavior for sub-word accesses. For example, during debugging we can program the PSTT of the HeapData checker (Figure 9) such that sub-word stores to uninitialized data leave the state of the word as uninitialized to detect all reads from uninitialized locations. In live runs, we can avoid false positives by programming the PSTT to treat a sub-word write as an initialization of the entire word.

When treating sub-word accesses as accesses to the entire word, our experiments did not have any false negatives (reads of uninitialized sub-words that might slip through without being noticed) because our benchmark suites, that are thoroughly tested and well structured, operate on data blocks that were multiples of word sizes. However, when sub-word accesses are treated in a paranoid fashion, i.e. when sub-word access does `not` set the state of the entire word to *Initialized*, numerous false positives are in many benchmarks, especially in the SPECint suite which frequently uses character and short integer arrays.

Alternatively, both false positives and false negatives on sub-word accesses can be avoided by i) Keeping state for each byte at a cost of increasing memory overhead needed for state. The same mechanisms described for word-level granularity still apply at byte-level tracking, or ii) Encoding the necessary semantic information as part of the state, like the solution proposed by [9].

3.2.6 Event Masking

It may be difficult to anticipate at compile time which checkers will be needed when the application is used. Therefore, it would be very useful to be able to switch different checkers on and off, depending on the situation in which the application runs. To achieve that, we can generate code with user events for several different checkers, and then provide a way to efficiently ignore events used for disabled checkers. This would also eliminate nearly all overheads when all checking is disabled.

To ignore load and store MemTracker events when checking is not used, we can set up the PSTT to generate no exceptions and no state changes. User events can be similarly neutralized, e.g. to turn the checker off without removing instrumentation for it. However, state and PSTT lookups would still affect performance and consume energy. To minimize this effect, we add an *event mask register* that has one bit for each type of event. If load and/or store events are masked out, loads and stores are performed without state and PSTT lookups. A masked-out user event instruction becomes a no-op, consuming only fetch and decode bandwidth.

3.2.7 Support for multiple checkers

As we have seen, MemTracker is an efficient hardware mechanism that can be programmed to implement different types of checkers. It can also be used to implement multiple checkers simultaneously. One way to do this is to directly merge state transition diagrams of the individual checkers into a combined state machine [76]. The advantage of this approach is that the total number of state bits in the combined checker can be fewer than the sum of state bits from the original checkers. For example, the three individual checkers used in our evaluation are HeapData, HeapChunks, and RetAddr and they use 4, 2, and 3 states, respectively (2, 1, and 2 bits of state per word). This results in a total of 5 bits of state per word (8 bit if only power-of-two state sizes are supported). However, when we design a checker to combine the functionality of these checkers, the resulting checker has only 7

states (3 bits of state per word, or 4 if only power-of-two state sizes are supported). The total number of states is reduced because some of the states can be eliminated as redundant, and the number of bits is further reduced because the individual checkers do not use all the states that can be encoded with the bits they use (e.g. the RetAddr checker uses two bits to encode only 3 states). This approach also has some disadvantages. In particular, we need a separate state diagram for each combination of checkers which does not facilitate using checkers as modules, requires extensive programmer effort when combining many checkers or checkers with many states, and it is difficult to add or remove a checker at runtime. The lack of checker modularity may also prevent simultaneous use of system and user checkers: the system may impose some checkers on an application (e.g. for security), and the application itself may want to use additional checkers (e.g. to detect and report bugs). In this scenario, the application-introduced checkers should not be able to affect (e.g. subvert) system-imposed checkers. This is difficult to achieve if a programmer must design the combined checker: the system programmer cannot anticipate all the application checkers that will be needed, whereas the application programmer cannot be trusted to design the combined checker because it subsumes the functionality of system-imposed checkers.

A more modular approach to combining checkers is to leave checkers independently defined. In general, a checker is expressed as a state transition function that takes the event ID and the current state, and returns the new state and an additional bit that indicates whether or not to raise an exception. If we have two checkers A and B, defined through functions F_A and F_B and using n_A and n_B state bits, respectively, the state of the combined checker will be a concatenation of individual checker's state bits, and the combined transition function will be a simple *dispatcher function* that calls F_A with the first n_A bits of the combined state, calls F_B for the second n_B bits of the combined state, concatenates the resulting state into a new combined state, and performs a logical *OR* of the individual exception bits to produce the exception bits of the combined checker. This is illustrated in Figure 10.

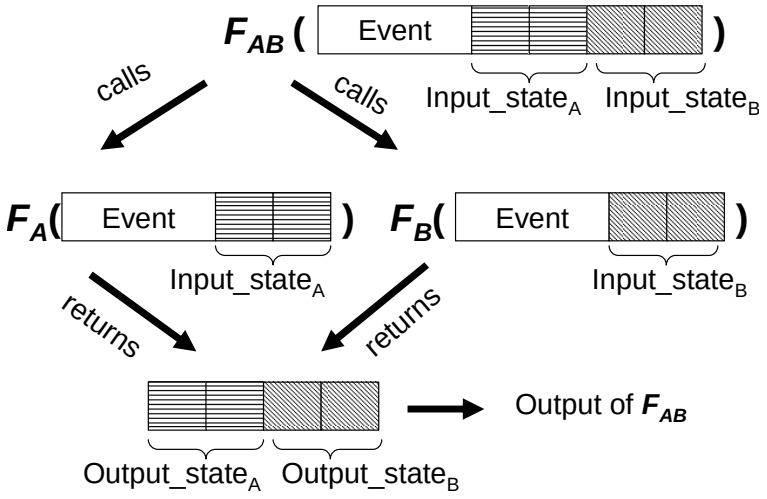


Figure 10: A dispatcher function F_{AB} calling F_A and F_B to determine the output of checkers A and B respectively.

With this approach, individual checkers can be specified as state transition functions, and the system can easily combine them without additional programmer involvement. Because each individual state transition function only sees its own part of the combined state, this also eliminates the risk of checkers interfering with each other. In particular, application-introduced checkers cannot affect the operation of system-imposed checkers.

It should be noted that the way checkers are combined affects the hardware design of MemTracker only indirectly, by making the state size larger and thus potentially requiring support for a larger maximum number of state bits. As far as MemTracker hardware is concerned, only one checker is used at a time, and the two approaches to checker combining only differ in how this one checker is constructed from component checkers.

Since the modular approach to checker combining can increase the number of state bits, we need to consider whether the the PSTT structure is capable of handling significantly larger numbers of states. With a maximum of four state bits (16 states), the number of PSTT entries is 576 (16 states and 36 events) and the entire PSTT is 360 bytes in size (576 entries, each with 4 bits of state and one exception bit). However, with the dispatcher implementation, we need a larger *maximum allowable number of state bits*, e.g. 8 or even

16 bits. With 8-bit state (256 states), the PSTT has 9216 entries and each entry is 9 bits in size (8+1), for a total PSTT size of slightly over 10KBytes. With 16 bit state, the PSTT size grows to nearly 5MBytes. While the original 360-byte PSTT can easily be stored on-chip and a state lookup can be performed quickly without hurting performance, this is unlikely to be the case with the larger PSTT needed to support more state bits.

To address this problem, we change the way the PSTT is designed. Instead of keeping the entire PSTT on-chip, we cache a few recently used PSTT entries on-chip, and use a software miss handler to service misses in this cache. This miss handler is effectively the state transition function F discussed earlier in this section - given the current state and event, it computes the new state and the exception bit and places this result in the PSTT cache to help avoid future PSTT cache misses. This approach also facilitates checker modularity and fast context switching - to change the current checker, the system must only invalidate the on-chip PSTT cache and change the address of the PSTT cache miss handler, instead of having to load the new content of the entire PSTT. This approach is similar to the technique used in FlexiTaint [74] for its taint propagation lookups.

3.3 Implementation

In this section, we outline the issues involved in implementing MemTracker in actual hardware. We first describe PSTT implementation and then show how MemTracker achieves efficient state lookups. We illustrate how MemTracker can be integrated into a modern out-of-order processor pipeline and also describe some features needed for multiprocessor implementation. Finally, we show how MemTracker interacts with operating system for setup and context switches.

3.3.1 Programmable State Transition Table (PSTT)

Different checkers that use MemTracker support need different numbers of state bits per word, so we provide support to select the number of state bits at runtime. In particular, we add a *MemTracker Control Register* (MTCR), which specifies the number of state bits per

word. Note that the number of state bits is determined by the checker and not by MemTracker. Hence the amount of virtual memory needed for storing state is also determined by MTCR. In our current implementation, we only allow power-of-two numbers, to simplify state lookups and updates. We also limit the maximum to 8 bits because the largest checker used in our evaluation (the modular combined checker) only uses 5 bits. Note that the number of supported state bits can be easily changed to 16 or even 32, at the cost of widening the MemTracker logic and PSTT cache entries. Note that with the new PSTT caching approach, we do not explicitly store the entire PSTT (it is expressed as a set of software handlers) so the larger number of bits does not result in exponential growth in storage needed to store the PSTT. In our current implementation, we use a small 256-entry PSTT cache structure. This small PSTT-cache is direct-mapped and is addressed using a concatenation of the event ID (6 bits to encode 36 events) and current state (8 bits in our implementation). Each entry consists of a tag (6 bits, 14 bits minus the 8 index bits), a new state (8 bits) and an exception bit (1 bit), for a total size of 480 bytes.

3.3.2 Finding State Information

State is stored in memory starting at the virtual address specified in the *State Base Address Register* (SBAR), and address of the state of a given data address can be found quickly by adding the SBAR with selected bits of the data address using simple indexing functions. An example lookup of 2-bit state for data address 0xABCD is shown in Figure 11.

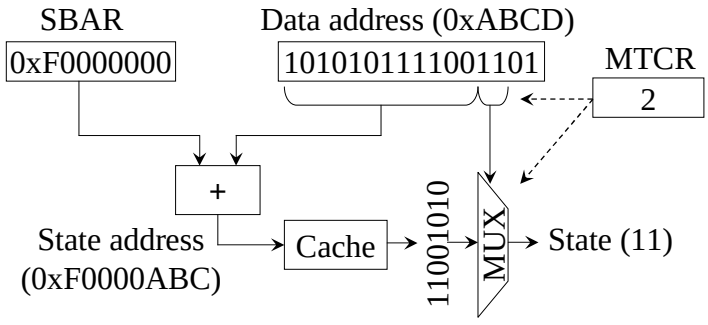


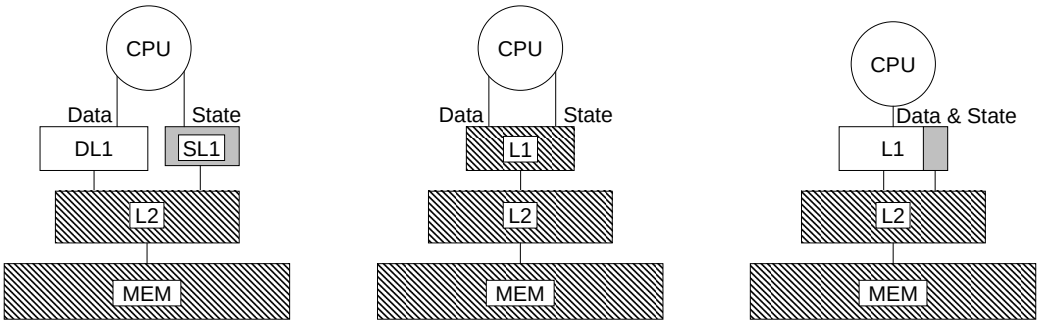
Figure 11: Lookup of 2-bit state for data location 0xABCD.

3.3.3 Caching State Information

There are three basic ways to cache state information: *split caching* (state blocks in a separate cache, Figure 12(a)), *shared caching* (state blocks share the same cache with data, Figure 12(b)), and *interleaved caching* (state bits stored with the cache line that has the corresponding data, Figure 12(c)). Shared caching has the advantage of using existing on-chip caches. However, in shared caching state competes with data for cache space, and lookups and updates compete with data accesses for cache bandwidth. Split caching has the advantage that it leaves the data cache unmodified, but adds extra cost for the separate state cache. Also, the state is usually much smaller than data. So, multiple states can fit in a single memory word and state accesses that exhibit good locality have higher hit rate in the state cache. Finally, interleaved caching allows one cache access to service both a data load and the lookup of the corresponding state; similarly, a data store and a state update can be performed in the same cache access. Conceptually, interleaved caching seems to be the logical choice because the state information is held as part of the same cache block[2]. For multiprocessors, maintaining atomicity of state and data is easier because, the state and data updates can be performed together. However, unlike the other two caching approaches, interleaved caching makes each cache block larger (to hold the maximum-size state for the block's data) and may slow down the cache even when state is not used (Table 2).

We find that the simple and inexpensive shared caching approach works well for non-primary caches. State is considerably smaller than the corresponding data, so the relatively few state blocks cause insignificant interference with data caching in large caches (L2 and beyond). Additionally, L1 caches act as excellent "filters" for the L2 cache, so state accesses add little contention to data block accesses. As a result, the choice of caching approach mainly applies to primary (L1) caches, and all three L1 caching approaches are

[2]This simplifies the lookup of the cache block (one lookup for both data and state) but still results in increased complexity because data and state may be in different parts of the same cache line.



(a) Split (b) Shared (c) Interleaved

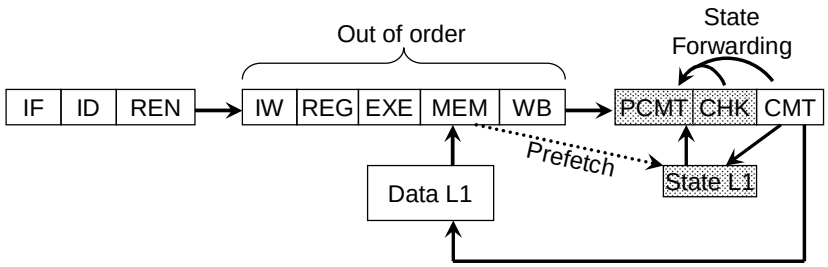
Figure 12: Caching approaches for MemTracker state in the L1 cache. For L2 and below we always use Shared.

examined in our experimental evaluation (Section 3.5). We find that shared L1 caching performs poorly without expensive additional cache ports. Interleaved L1 caching performs slightly better at an added cost of access latency and area, but it simplifies some memory consistency issues (Section 3.3.6) and may still be good choice in chip-multiprocessors. Finally, split caching has good performance even with a smaller and simpler state cache, and has a good performance-cost-power trade-off.

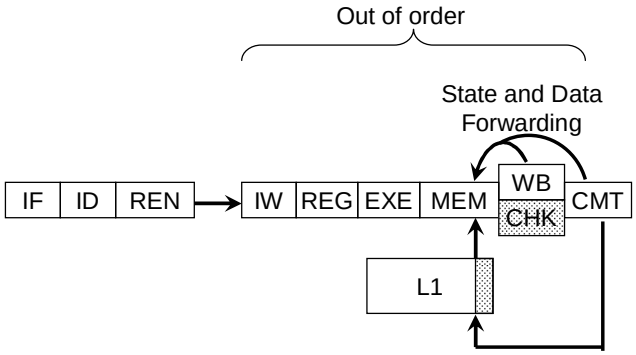
3.3.4 Processor Modifications for MemTracker

MemTracker integration into the processor pipeline is simpler for Split and Shared L1 state caching approaches, where state lookups can be delayed until the end of the pipeline. This allows MemTracker to be added as an in-order extension to the commit stage of the pipeline, avoiding any significant changes to the complex out-of-order engine of the processor (Figure 13(a)).

In a conventional processor (no MemTracker support), the commit logic of the processor checks the oldest instructions in the ROB and commits the completed ones in order (from oldest to youngest). If an instruction is a store, its commit initiates the cache write access. These writes are delayed until commit because cached data reflects the architectural state of memory and should not be polluted with speculative values.



(a) Shared and Split



(b) Interleaved

Figure 13: Processor pipeline with MemTracker support (shaded) for different L1 state caching approaches.

MemTracker adds two additional pipeline stages just before the commit (Figure 13(a)). The first of these stages is the *pre-commit stage* (PCMT), which checks the oldest few instructions in the ROB and lets the completed instructions to proceed in-order into the next pipeline stage. For MemTracker events (loads, stores, and user events) pre-commit also fetches MemTracker state from the state cache. In the event the state lookup results in a cache miss, the pipeline is stalled until the miss is serviced. Because MemTracker processes instructions in-order, a cache miss here will stall the commit of instructions, which can be costly. We alleviate this problem by issuing a non-binding state prefetch when the data address is resolved. This prefetch is dropped if no cache port is available, to avoid contention between these prefetches and state lookups (or with data accesses in the shared configuration). Our experiments indicate that this state prefetching is highly effective, it eliminates nearly all state misses in the pre-commit stage without the need to add any additional cache ports.

In the second MemTracker pipeline stage (*check stage* or CHK), the state and the event type are used to look up the corresponding PSTT cache entry. If the state is not available (state cache miss), the resulting exception is treated as any other exception: it causes the instruction (and all instructions fetched after it) to be squashed, the miss handler is called (it computes the PSTT entry and updates the PSTT cache), and after the handler completes the squashed instruction is re-executed (it now finds the PSTT entry in the PSTT cache and completes). An exception also occurs if the PSTT indicates that an exception should be raised: the current instruction and all younger instructions are squashed and the processor begins to fetch instructions from the exception handler.

If there is no exception, the instruction proceeds to the commit stage. If the new state from the PSTT is different from the current state, the state is written to the state L1 cache at commit, at the same point when stores normally deposit their data values to the data cache.

State checks in the check stage can have dependencies on still-uncommitted state modifications, so a small state forwarding queue is used to correctly handle such dependence. This is similar to the "conventional" store queue which forwards store values to dependent loads, but our state forwarding queue is much simpler because i) it only tracks state updates in the two stages between pre-commit and commit, so in a 4-wide processor we need at most 8 entries, and ii) all addresses are already resolved when instructions enter this queue, so dependencies can always be precisely identified. If multiple store instructions are to forward the state to their dependent instructions in the same clock cyle, such forwarding is done one at a time (potentially increasing the execution time). This is done to reduce the complexity of the forwarding logic.

In the interleaved state caching approach, the main advantage of interleaving is to have a single cache access read or write both data and state. As a result, state lookups are performed as soon as the data (and state) address is resolved (MEM stage in Figure 13(b)). To achieve this, MemTracker functionality is weaved into the "conventional" processor

pipeline and there are no additional pipeline stages. State lookups and updates are performed in much the same way as data loads and stores: lookups when the address is resolved and updates when the instruction commits. Consequently, state must be forwarded just like data, and speculative out-of-order lookups must be squashed and replayed (such squashes are rare , and none were observed in any of our experiments) if they read state that is later found to be obsolete. As a result, the state lookup/update queues in this approach are nearly identical to load/store queues in functionality and implementation, but are less complex and power-hungry because state is much (by a factor of 4 or more) smaller than the corresponding data. Finally, it should be noted that, if load/store queues are replaced in the future by some other forwarding and conflict resolution mechanism(s), our state lookup/update queues can be replaced by the same forwarding and conflict resolution mechanism(s).

3.3.5 Filtering of Silent State Writes

A store instruction, in addition to performing data write, now has to fetch the state and possibly update the state. Hence every store instruction could have up to three memory accesses. This is expensive especially for the state cache that could have two accesses (one read and one write) every time a store is performed. Because every state update is preceded by a state lookup, silent updates can easily be detected (the new state is the same as the old one) and eliminated. This saves contention for cache ports and reduces extra energy consumption by avoiding unnecessary cache accesses. In multiprocessor configurations, the elimination of a state write also eliminates the need to enforce write atomicity (this will be explained in Section 3.3.6. We observe in our experiments that this optimization eliminates 98.5% of state writes on average across different benchmark suites.

3.3.6 Multiprocessor Consistency Issues

MemTracker states are treated just like any other data outside the processor and its L1 cache, so state is automatically kept coherent in a multiprocessor system. Hence, we focus

our attention on memory consistency issues. We use the strictest consistency model (sequential consistency) in our implementation of MemTracker. We also briefly explain how to support processor consistency, on which many current machines are based.

Because MemTracker stores state separately from data in memory and L2 caches, the ordering of data accesses themselves is not affected. The ordering of state accesses can be kept consistent using the same mechanisms that enforce data consistency. However, MemTracker introduces a new problem of ensuring that the ordering between data and state accesses is consistent. In particular, even in a RISC processor, a single load instruction could result in a data read, a state lookup, and a state update; similarly, a store instruction could result in a state lookup, a data write, and a state update. In a sequentially consistent implementation, data and state accesses from a single instruction must appear atomic. This creates three separate issues: atomicity of state and data writes in store instructions, atomicity of state and data reads in load instructions, and atomicity of state reads and writes in all event instructions.

Atomicity of state and data writes in a store instruction is easily maintained in interleaved caching because the same cache access writes both data and state, hence the two writes are actually simultaneous. In split and shared caching, we force both writes to be performed in the same cycle. If one suffers a cache miss, the other is delayed until both are cache hits and can be performed in the same cycle.

Atomicity of the state lookup and the data read in a load instruction is also easily maintained in interleaved caching, because they are again part of the same cache access. In split and shared caching, the instruction must be replayed if the data value has (or may have) changed before the state is read by the same instruction. For this, we can use the processor's existing mechanism for enforcing load-load ordering. In most processors, this involves replaying the instruction when an invalidation for the data block is received before the instruction commits. Other processors can have a different mechanism to trigger load-load replay mechanism, e.g. check if the data value read at commit differs from the one

originally loaded from the cache. If the state access triggers a page fault, it is treated as any other page fault: the instruction is canceled, the page fault is serviced, and the execution of the application starts by re-executing the faulting instruction.

Finally, atomicity of the state lookup and update can be maintained using the same replay mechanism, i.e., the instruction is replayed if the state read by the instruction changes before the instruction commits its state change to the cache.

In consistency models that allow write buffers (e.g., processor consistency), the simplest way to ensure correct behavior is to flush the write buffer when there is a state update and do the write directly to the cache. This approach should work well when state updates are much less frequent than data writes, as we show in Section 3.5.

Overall, MemTracker support can be correctly implemented in sequentially and processor-consistent multiprocessors in a relatively straightforward way, by extending existing approaches that maintain data consistency. We note that any mechanism that maintains state (e.g. fine-grain protection) separately from data would have similar issues and demand similar solutions.

3.3.7 Setup and Context-Switching

The integrity of MemTracker-related information such as PSTT-cache or PSTT (in our original implementation) can be compromised if adequate steps are not taken to prevent users from corrupting the data in the transition table. We maintain PSTT-related information as part of the process context that cannot be modified directly by the application. Initializing the PSTT-cache miss handler register or the PSTT (in our prior implementation) is performed through a system call before loading the application. The Operating System is responsible for initializing the PSTT structures in privileged mode and keeping the PSTT in system memory that is not mapped into the application's address space. This prevents the users from directly modifying the PSTT.

As we consider MemTracker-related processor state (PSTT-cache, PSTT cache miss

handler exception address register, MTCR, event mask register) to be a part of process state, this information is saved and restored on context switches. In our original implementation [76], the total amount of MemTracker state to be saved/restored in this manner was nearly 300 bytes because the entire PSTT was saved and restored. In our new implementation, the entire PSTT is not actually stored anywhere - its entries are dynamically generated as needed. Thus, the PSTT cache can simply be invalidated on a context switch, and the saved/restored MemTracker state consists of only the three registers listed above. Note that per-word states need not be saved/restored on context switches. Instead, they are only cached on-chip and move on- and off-chip as a result of cache fetch and replacement policy.

3.4 Evaluation Setup

In this section, we describe the different memory checkers and benchmarks that we use in our evaluation and show our simulation environment.

3.4.1 Memory Problem Detectors

To demonstrate the generality of our MemTracker support and evaluate its performance more thoroughly, we use four checkers designed to detect different memory-related problems. One of these checkers is the *HeapData* checker used as an example in Section 3.2.1.

Event / State	UEVT30 (SetDelimit)	UEVT31 (ClrDelimit)	LD	ST	Sub-word LD	Sub-word ST
0 (Normal)	1	0	0	0	0	0
1 (Delimit)	1 E	0	1 E	1 E	1 E	1 E

Figure 14: PSTT setup for the *HeapChunks checker.*

The second checker is *HeapChunks*, which detects heap buffer overflows from sequential accesses. For this checker, the memory allocation library is modified to surround each allocated data block with delimiter words, whose state is changed (using event UEVT30) to *Delimit*, while all other words remain in the *Normal* state. Any access to a *Delimit* word

is an error. When the block is freed, the state of its delimiter words is changed back to *Normal* (using UEVT31). Note that the standard GNU heap library implementation keeps meta-data for each block (block length, allocation status, etc.) right before the data area of the block. As a result, we do not need to allocate additional delimiter words, but rather simply use these meta-data words as delimiters.

This HeapChunks checker uses one-bit state per word, and the PSTT for it is shown in Figure 14. Note that HeapChunks is intended as an example of a very simple checker with one-bit state, and that it does not provide full protection from heap block overflows. For example, it would not detect out-of bounds accesses due to integer overflow or strided access patterns. However, it does detect the most prevalent kind of heap-based attacks (sequential buffer overrun).

Event / State	UEVT24 (RAwr)	UEVT25 (RArd)	UEVT26 (RAfree)	LD	ST	Sub-word LD	Sub-word ST
0 (NotRA)	1	0 E	0 E	0	0	0	0
1 (GoodRA)	1 E	1	0	1	2	1	2
2 (BadRA)	1	2 E	0	2	2	2	2

Figure 15: PSTT setup for the *RetAddr checker.*

The third checker is *RetAddr* (Figure 15), which detects when a return address on the stack is modified. This checker detects stack smashing attacks that attempt to redirect program control flow by overwriting a return address on the stack. This checker keeps each word in the stack region in one of three states: *NotRA*, *GoodRA*, and *BadRA*. All stack words start in the *NotRA* state, which indicates that no return address is stored there. When a return address is stored in a stack location, its state changes to *GoodRA*. An ordinary store changes this state to *BadRA*. When a return address is loaded, we check the state of the location and trigger an exception if the location is not in the *GoodRA* state. Our simulations use the MIPS ISA, which uses ordinary load and store instructions to save/restore the return address of a function, which is otherwise kept in a general-purpose register. To

expose return address activity to our checker, we insert UEVT24 (*RAwr*) after each valid return address store, UEVT25 (*RArd*) before each valid return address load, and UEVT26 (*RAfree*) before the return address stack location goes out of scope (when the activation record for the function is deallocated). All these user events target the intended location for the return address. For our experiments, this event insertion is done by a modified GCC code generator, but it would be relatively simple to achieve the same goal through binary rewriting. For CISC processors (e.g. x86), return address pushes and pops are done as part of function call/return instructions, so it is trivial to identify them. The *RetAddr* checker is another example of a useful checker that can benefit from MemTracker due to frequent state updates: each function call/return will result in at least three state updates to the return address' state.

The fourth checker combines all three checkers described above. We show two different implementations of this combined checker as discussed in Section 3.2.7. Our first implementation uses seven different states that implements all three checkers in a single checker, and configures MemTracker to use four-bit states. This implementation uses minimal number of state bits. Our second implementation uses a dispatcher to internally call the individual checkers and uses eight state bits (two for HeapData, two for RetAddr checker and one for HeapChunks padded with three bits to have a power-of-two number of state bits). The dispatcher implementation of the combined checker is the most demanding of the four checkers in terms of the number of user events that must be executed and in terms of state memory and on-chip storage requirements, so we use it as the "default" checker in our evaluation and use the three component checkers to evaluate the sensitivity of MemTracker's performance to different checkers and state sizes.

3.4.2 Benchmark Applications

We use all applications from the SPEC CPU 2000 and 24 out of 29 applications for SPEC CPU 2006 application suite [67] benchmark suite. For each application, we use the reference input set in which we fast-forward through the first fifteen percent of instructions to skip initialization phases and simulate the next one billion instructions in detail. We note that our fast-forward must still model all MemTracker state updates to keep the checker's state correct. If we ignore allocations, initializations, and return address save/restore while fast-forwarding, when we enter detailed simulation, our checkers would trigger numerous exceptions due to falsely detected problems (e.g. reads from locations whose allocation we skipped). We also evaluate Splash-2 benchmark suite [78] for our multiprocessor evaluation. We run these applications from start to end.

3.4.3 Simulation Environment and Configuration

We use SESC [57], an open-source execution-driven simulator, to simulate a MIPS processor. We derive microarchitectural parameters from a modern processor (Intel Core2-like) running at 2.93GHz. The L1 data cache we model is 32KBytes in size, eight-way set associative, dual-ported (2 RW ports), with 64-byte blocks with a hit latency of 2 cycles and miss latency of 1 cycle. The L2 cache is 4MByte in size, sixteen-way set associative, single-ported, and also with 64-byte blocks. Hit latency for L2 caches is 10 cycles and miss latency is 4 cycles. The processor-memory bus is 64 bits wide and operates at 1333MHz. The round-trip memory latency is assumed to be 490 cycles. For multiprocessor evaluation, we use a four core configuration with one L1 cache per core and a shared L2, using the same cache parameters as in our uniprocessor simulations.

Our default MemTracker configuration (shown in black in all charts) uses split state caching in the L1 cache (Figure 12(a)), with 16KBytes of state cache, which is four-way set-associative, dual-ported, and with 64-byte blocks.

3.5 Evaluation

In this section, we conduct experiments to evaluate MemTracker. We first show the effect of different L1 caching approaches described in Section 3.3.3. Second, we show the performance overheads of different checkers that we described in Section 3.4.1. Third, we conduct sensitivity analyses with different L1 State Cache sizes. We then compare our technique with prior mechanisms. Finally we show latency, area and energy overheads of MemTracker system and conduct validation experiments to verify the correctness of our implementation.

3.5.1 Effect of L1 Caching Approaches

As described in Section 3.3.3, we examine three different approaches to caching state in primary caches: *Split*, *Shared*, and *Interleaved*. Figure 16 shows execution time overheads for a representative set of benchmarks along with averages across entire benchmark suites on the most demanding *Combined* checker with 8 bits of state, relative to a system without any checking and without MemTracker support. We observe that the *Split* configuration has a performance overhead of around 2.8% average across SPEC benchmarks and 3.15% average across SPLASH-2 benchmarks with the worst-case (around 14.5%) in art benchmark, with a relatively smaller 16KByte L1 state cache. The lower-cost *Shared* approach exhibits consistently higher overheads on average, and its overhead also varies considerably across benchmarks, with the worst case around 14.7% (in art). The higher overheads are caused by contention between state and data for both L1 cache space and bandwidth. The only advantage of the *Shared* approach is reduced cost due to using the existing L1 cache, it makes little sense to add L1 ports or capacity to reduce this overhead - an additional port would make 32KByte L1 data more power hungry and a larger L1 cache would increase latency.

Finally, the *Interleaved* approach has almost similar overheads as the *Split* configuration. This configuration has dedicated space for state in each L1 line, so this configuration

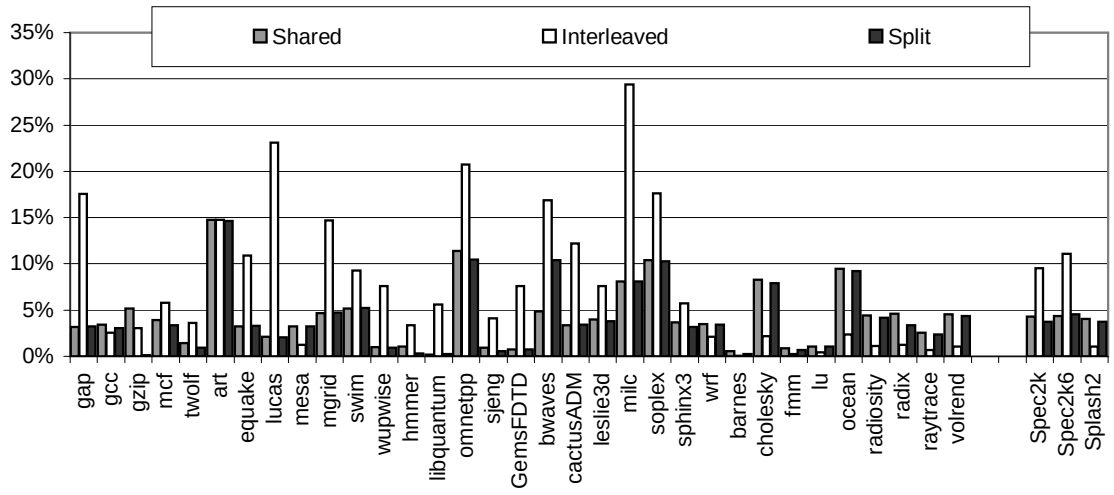


Figure 16: Effect of shared, split, and interleaved caching of state and data in L1 caches.

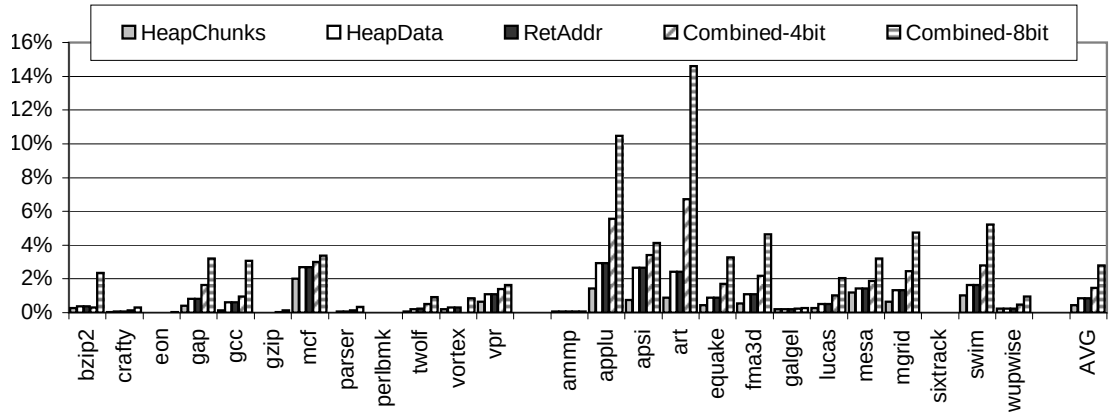
will exhibit the same performance overheads even when MemTracker is turned off and will leave the cache slower and more power hungry due to additional bits. Further analysis results are presented in Section 3.5.5.

Overall, the *Split* configuration has the relatively better average performance-cost trade-off than other configurations. It is also comparatively easy to integrate into the processor pipeline using the in-order pre-commit implementation described in Section 3.3.4. The additional 16KByte L1 state cache in this configuration is only half the size of the L1 data cache, and adds little to the overall real-estate of a modern processor. Hence, we use this *Split* configuration as the default MemTracker setup in the rest of our experiments. However, we note that the *Interleaved* approach has some advantages in terms of multi-processor implementation (Section 3.3.6) and, with additional L1 bandwidth, has similar performance to the *Split* configuration. Consequently, the *Interleaved* approach with added L1 cache bandwidth may also be a good choice if the goal is to simplify support for multi-processor consistency.

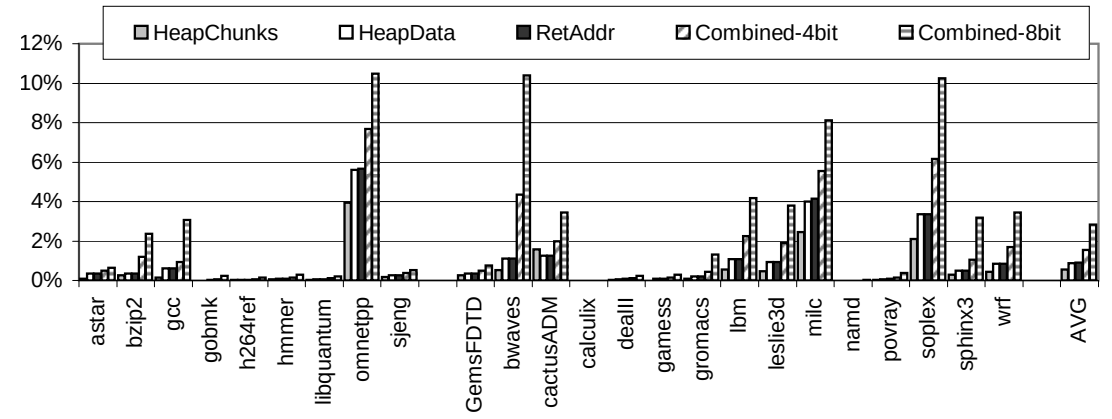
3.5.2 Performance with Different Checkers

Figure 17 shows that the overhead mostly depends on the number of state bits per word used by a checker. The one-bit *HeapChunks* checker has the lowest overhead – around 0.5% on average and 2.45% worst-case in *milc* benchmark. Both two-bit checkers have similar overheads of around 1% on average and 4% worst-case for HeapData and 4.15% for RetAddr checker both in *milc* benchmark. We note that these checkers have different user events – the *HeapData* checker uses variable-footprint user event instructions to identify heap memory allocation and deallocation, while the *RetAddr* checker uses word-sized user events to identify when the return address is saved and restored. However, after application's initialization phase, *HeapData*'s user events are not frequent, while each of *RetAddr*'s more-frequent events requires little processing. As a result, in both checkers user events contribute little to the overall execution time.

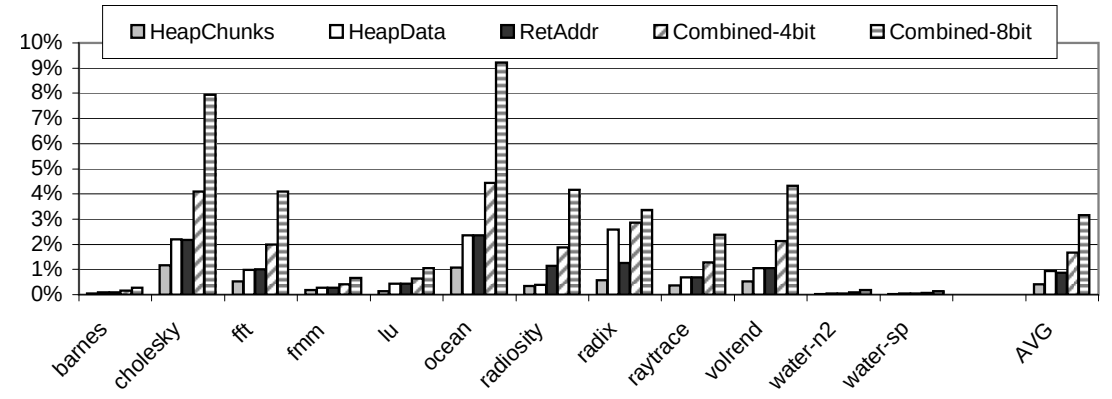
The four-bit *Combined* checker has an overhead of 1.5% on average and about 6.7% worst-case (in art). This overhead is larger than in less-demanding checkers, mainly due to larger state competing with data for L2 cache space and bus bandwidth. Still, even the "high" 6.7% worst-case overhead is low enough to allow checking even for "live" performance-critical runs. Also, note that the overhead of the combined checker is significantly lower than the sum of overheads for its component checkers. This is mainly because the combined checker does not simply do three different checks – they are combined into a single check. Finally, as expected, the eight-bit *Combined* checker exhibits slightly worse overheads than the more efficient four-bit combined checker. The averages are around 2.8% for SPEC benchmarks and 3.15% for SPLASH-2 benchmarks with the worst case of around 14.5% in art benchmark. These higher overheads are largely due to higher contention for capacity and bandwidth for L2 cache shared by both data and state. We use 256 entry PSTT-cache in our experiments. We did not notice any significant overheads due to misses in PSTT cache beyond the initial cold misses. However, we note that, as more sophisticated checkers are implemented, there may be an increased demand for



(a) Spec2000 benchmarks



(b) Spec2006 benchmarks



(c) Splash-2 benchmarks

Figure 17: Overhead of different checkers in MemTracker, with Split L1 state caching using a 16KByte L1 state cache.

PSTT-cache entries and hence, a larger PSTT might be needed.

3.5.3 Sensitivity Analysis

We performed additional experiments with 2KByte, with 4KByte, with 8KByte and with 32KByte (all with 4-way associativity and 64 byte line size) state L1 state caches in the *Split* configuration in addition to our default 16KByte L1 state cache used in our experiments. Figure 18 shows the results of these experiments for a subset of the benchmarks along with averages across entire benchmark suites. We find that state caches larger than our default of 16KBytes bring negligible performance improvements (<0.5%) in all applications and on average, which indicates that the 16KByte cache is large enough to capture most of the first working set for MemTracker state. The smaller 8KByte cache still shows almost similar overheads for all benchmarks except ocean benchmark where an 8KByte cache has higher miss rate than the 16KByte. For smaller cache sizes namely 4KByte and 2 KByte, the overheads progressively worsen due to higher contention for cache capacity. The reason for this is that the smaller state cache "covers" less memory than the L1 data cache for the 8-bit Combined checker used in these experiments, which puts a larger number of state L1 cache misses on the critical path. Additionally, line size in the state cache is the same as in the data cache (64 bytes in our experiments) although state is smaller than the corresponding data. This puts smaller state caches at a disadvantage in applications with less spatial locality. While some applications like mgrid, bwaves, sphinx3 and ocean show that performance can be progressively improved with higher capacity state caches, certain application like applu, art, soplex are agnostic to increasing cache sizes. On further investigation, we find that these applications have increased contention on L2 bandwidth caused by the sharing between data and state information. However, on an average we find that caches with higher capacity tend to improve performance overheads across benchmarks.

We also conducted experiments in which we disable state prefetches (see Figure 13(a)) in the *Split* configuration. We find that the average overhead increases by around 2.7%

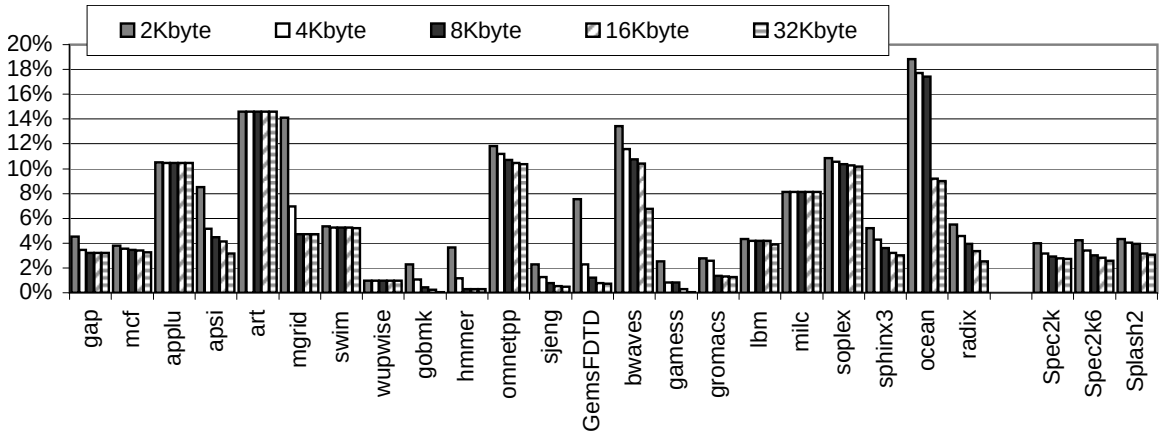


Figure 18: Performance overhead variation due to different sizes of L1 state cache.

without state prefetching. Our state prefetching mechanism is very simple to implement, and we believe its implementation is justified by the reduction in both average overhead and variation of overheads across applications.

Overall, we find that the 16KByte state cache results in a good cost-performance trade-off, and even though smaller state caches can be used to reduce cost if a wider variation in performance overhead and slightly higher average overheads are acceptable. We also find that state prefetching brings significant benefits at very little cost.

3.5.4 Comparison with Prior Mechanisms

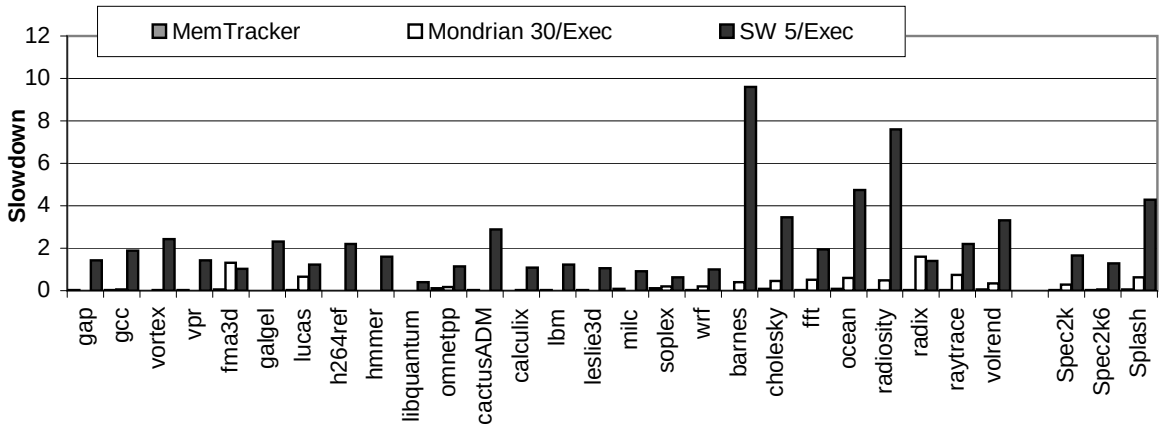


Figure 19: Effect of state changes in software handlers.

To estimate the advantages of our MemTracker support, we compare its performance with checking based on an approximation of Mondrian Memory Protection [77] and with an approximation of checking based on software-only instrumentation. It should be noted that Mondrian was not intended to be a general purpose tagging scheme and our experiments do not show how Mondrian performs poorly for its intended purpose (fine grain protection). Instead, we use Mondrian to show that fine-grain memory protection cannot be efficiently used to emulate MemTracker. We do not actually implement these schemes, but rather make optimistic estimates of the cost for key checking activities. As a result, these estimates are likely to *underestimate* the actual advantages of MemTracker over prior schemes, but even so they serve to highlight the key benefits of our mechanism.

In a Mondrian-based implementation of the Combined checker, Mondrian's fine-grain permissions must be set to only allow accesses that are known to be free of exceptions and state changes. Examples of such accesses are load/store to already initialized data, load/store to non-return-address stack locations, and loads from unmodified return address locations. We assume zero overhead for these accesses, which makes Mondrian permission fetches and checks "free". For permissions changes on allocation, deallocation, or return address load/stores, we model only a penalty of 30 cycles for raising an exception. We note that this penalty is optimistic, because it subsumes a pipeline flush for the exception, the jump to the exception handler, the actual execution of the exception handler (which must update Mondrian's permissions trie structure), and the return to the exception site.

For software-only checking, we only model a five-cycle penalty for each check that must be performed for a load/store. This check must read the state for the target memory location, determine if a state change is needed or if an error is indicated, and finally update the state. The 5-cycle penalty is added to the execution time of the unmodified application, so the penalty includes all effects of instrumentation, including any misses in the instruction cache, misses in the data cache when looking up state, conditional branches when checking the state, and actual execution of instrumentation instructions. We note that

this 5-cycle penalty is very optimistic; for example, in HeapMon [62] the actual reported average duration of a (highly optimized) load or store check is 18 to 480 cycles, depending on the application.

The results of these experiments are shown in Figure 19. For MemTracker, we use a Split L1 caching configuration with a 16KBytes L1 state cache. We find that our MemTracker mechanism (with all overheads accounted for) outperforms both Mondrian-based checking and software-only checking. The average slowdown for software-only checking is 1.6× (times) on average across SPEC-2006 benchmarks, 1.3× across SPEC-2000 benchmarks and 4.3× for SPLASH-2 applications, with a worst-case of 9.6× for barnes. Due to our optimistic assumptions for software-only checking, this overhead is lower than previously reported numbers [50, 73] for such checkers, but it is still too high to allow always-on checking in production runs.

The average overhead for Mondrian-based checking is 29% average across SPEC-2006 applications, 6% average across SPEC-2000 benchmarks and 63% across SPLASH-2 applications. Many applications in the Mondrian-based scheme behave similar to MemTracker as the number of state changes in these applications are relatively few. However, certain applications like fma3d, lucas, barnes, cholesky and radix have much higher overheads due to higher frequency of state changes. Since MemTracker can perform such state changes automatically in hardware while a Mondrian-based scheme need to invoke a software handler (assumed to have a 30-cycle penalty), they result in large overheads for some benchmarks. We note that Mondrian requires complex hardware to look up and manage its trie permissions structures and uses several kinds of on-chip caching to speed up its permissions checks. As a result, Mondrian implementation is unlikely to be less complex than MemTracker, so MemTracker's higher performance and lower performance variation across applications is a definite advantage. It should also be noted that we fully model all overheads for MemTracker-based checking, whereas the real overheads of Mondrian-based checking could be considerably higher than our optimistic estimate.

Table 2: Latency, area, and power overheads, expressed as a fraction of the latency, area, and power of the unmodified 32 KByte L1 cache.

	Latency	Area	Power
Split	0.0%	56.9 %	21.9%
Shared	0.0%	0.0%	101.5%
Interleaved	21.4%	25.1%	43.0%

3.5.5 Latency, area and power overheads

We perform experiments to determine the latency, area and power overheads due to different caching configurations namely *Split*, *Shared* and *Interleaved*. We use Cacti 4.2 [37], an integrated access time, area and dynamic power model to model overheads due to our different caches. The results are shown in Table 2.

The *Split* configuration stores the state information in a separate but smaller cache. This allows the latency of accessing the state to be hidden by the longer latency of the L1 data cache, but the additional state cache occupies area equal to approximately 57% of the size of the existing L1 data cache. In modern processors, it is estimated that only about 2.9% of total on-chip area dedicated to L1 Data cache [45], so the contribution of the state L1 cache to the overall chip area is small (approximately 1.7%). The *Shared* cache configuration does not add latency or area to the L1 data cache; however, every memory access involves an extra access for state information (which doubles the energy cost) and a small percentage of memory access have a third access to the data cache to update the state information. This requires additional L1 ports and *more than doubles* the energy of accessing the L1 cache. For *Interleaved* caching configuration, the cache lines are extended to accommodate the 8 bit states corresponding to each memory word. This increases the L1 cache access latency by 21.4% even when MemTracker is turned off. Also, the area of an L1 data cache increases by a quarter of its original size. While all loads will fetch the state information along with data, stores have to perform state checks before actually performing writes to memory. Additionally, there will be a small percentage of time when

a state update is needed. Hence the dynamic power costs grows by almost 43% for the interleaved configuation.

3.5.6 Validation of Access Checking Functionality

We tested our checking functionality by injecting bugs and attacks into several applications as they are running with our *Combined* checker. All instructions of the applications are simulated from the start of execution until either a bug/attack is detected, or until they complete execution, in which case we check the program's results for correctness.

To test the return address protection, we choose 15 different function calls in each of the following applications: crafty, parser, and twolf. After the return address is saved to the stack, we inject a single dynamic instance of a store instruction that overwrites it. Our checker detects all such attacks, raising an exception before the modified return address is actually used to re-direct control flow of the application.

To test the heap chunk protection, we randomly choose an allocated heap block and sequentially overwrite the block past its end. We performed a total of 60 such attacks in crafty, gzip, mcf, and mesa, and our checker always detects the write that exceeds the allocated space.

To test our detection of reads from uninitialized heap locations, we randomly choose a dynamic instance of a `calloc` call and omit the initialization of the first or the last word of the block. We injected a total of 122 such errors in crafty, gzip, mcf, and mesa, and in all but one injection, reads from the uninitialized location were detected. The remaining one injection (in gzip) was not detected because the application never read the word whose initialization we omitted.

Finally, to test our detection of accesses to unallocated heap data, we intercept a randomly-chosen dynamic instance of a `malloc` call and reduce the size of the request by 4 bytes (one word). We performed a total of 183 such injections in crafty, gzip, mcf, and mesa. In 149 of these injections our checker finds a read or a write to the unallocated location. In

the remaining 34 injections the last word of the allocated block is never actually accessed, so the injected error is not manifested (the application completes correctly).

Although our checkers are very effective in finding the errors and attacks they target, we note that the checkers themselves are not the focus of our work. They are only used to demonstrate and test our MemTracker mechanism, and problem detection abilities of these checkers are similar to other implementations of similar checkers.

3.6 Related Work

The most generic of the previously proposed hardware mechanisms is DISE [15] (Dynamic Instruction Stream Editing), which pattern-matches decoded instructions against templates and can replace these instructions with parameterized code. For memory access checking, DISE provides efficient interception that allows instrumentation to be injected into the processor's instruction stream. In contrast to DISE, MemTracker does not modify the performance-critical front-end of the pipeline, and it performs load/store checks without adding dynamic instructions to execution.

Horizon [40] widens each memory location by six bits, two with hard-wired functionality and four trap bits that can intercept different flavors of memory accesses. Mondrian Memory Protection [77] has per-word permissions that can intercept read, write, or all accesses. iWatcher [82] provides enhanced watchpoint support for multiple regions of memory and can intercept read, write, or all accesses to such a region. HeapMon [62] intercepts accesses to a region of memory and uses word-granularity filter bits to specify locations whose accesses should be ignored, with the checker implemented as a helper thread. All four schemes only provide interception and state checking in hardware. Each state update requires software intervention to change trap, permission, watchpoint, or filter bits (in Horizon, Mondrian, iWatcher, and HeapMon, respectively). This software intervention can take the form of instrumentation at the point where the change is needed. Alternatively, accesses that require state change can result in an exception, allowing the exception

handler to change the state. For example, consider a state that indicates whether a location is initialized. Initialized locations can have their trap, permission, watchpoint, or filter bits set to indicate that both reads and writes are allowed without generating exceptions. Uninitialized locations should have indicate that both reads and writes require exceptions, reads to indicate an error (read from uninitialized location is detected) and writes to allow the exception handler to change the state to "initialized". In contrast to these schemes, MemTracker can be programmed to handle state changes in hardware, without software intervention. This is an important difference because, as will be shown in Section 3.5.4, in some useful checkers, state changes are numerous enough to cause significant overheads.

To avoid hard-wiring the number of bits for each state, but still provide efficient checks and updates, MemTracker uses a flat (array) state structure in memory. In contrast, Mondrian [77] uses a sophisticated trie structure to minimize state storage overheads for coarse-grain state, at the cost of more complex fine-grain state updates. iWatcher [82] keeps track of ranges of data locations with the same state, which also complicates fine-grain updates. Horizon [40] simplifies state lookups by keeping state in six extra bits added to each memory location, which requires non-standard memory modules and adds a state storage cost even when no checks are actually needed. In contrast, MemTracker keeps state information separately in ordinary memory, and uses only as much state as needed. In particular, when checking is not used, there is no memory allocated for MemTracker state.

Hardbound [20] adopts decoupling data and state storage used by MemTracker. HardBound is hard-wired for a particular scheme that provides spatial safety guarantees by storing bounds information for pointers separately and keeping it transparent to the underlying hardware. Since the meta-data is stored in the pointer itself, it is difficult to find accesses that use stale or dangling pointers. In contrast, MemTracker associates state with memory words themselves. After memory is freed, the state associated with those memory words would reflect that the memory is "free" and an access to that memory through a dangling pointer can be detected correctly.

Nethercote et al. [49] have done detailed studies for shadow memory management needed by software memory checkers such as Valgrind. Their design allocates a byte of meta-data for every byte in memory and proposes techniques for optimizing the memory overhead. We expect the number of state bits for every memory word to be typically small (maximum of 8 or 16 state bits per word) as evidenced in our evaluation (See Section 3.5). Hence, we do not anticipate the amount of virtual memory needed for state to be be a limiting problem especially for larger address spaces as we move from 32-bit to 64-bit addresses. Also, the optimizations such as compression employed by [49] can be adopted by MemTracker to support more sophisticated checkers that require larger percentage of virtual memory space.

Other related work includes AccMon [81], Dynamic information flow tracking [68], Minos [17], Memory centric security [65], LIFT [56], WIT [1], LBA [9] and SafeMem [55]. AccMon uses "golden" (correct) runs to learn which instructions should access which locations, then checks this at runtime using Bloom filters to avoid unnecessary invocations of checker code; Dynamic information flow tracking and Minos add one *integrity* bit to each location to track whether the location's value is from an untrusted source; SafeMem scrambles existing ECC bits to trigger exceptions when un-allocated locations are accessed and to help garbage-collection. LIFT does dynamic software instrumentation and relies heavily on optimized code to be inserted for checks. WIT uses points-to analysis at compile time and inserts guards around objects to detect buffer overflow attacks. Memory Centric architecture associates security attributes to memory instead of individual user process. Most of the above mechanisms are designed with specific checks in mind: in AccMon, much of the hardware is specific to its heuristic-based checking scheme; in Dynamic information flow tracking and Minos, the extra bit tracks only the integrity status of the location; SafeMem cannot track per-word state and can only intercept accesses to blocks with no useful values – a block with useful values needs a valid ECC to protect its values from errors. LBA

adopts some of the key contributions of MemTracker such as hardware support for accelerating memory checkers and programmability to suport multiple checkers. Additionally, it proposes sophisticated (and more complex) address translation mechanisms for state. MemTracker relies on simple indexing functions for state lookups to avoid performance loss. Also, LBA implements variable state granularity (e.g., per-byte, per-word etc.) and supports state semantics (e.g., encoding lock-set information for memory words).

Overall, MemTracker is unique in that, it can efficiently support different memory checkers, even those that require simple but frequent state updates automatically without software intervention. It should be noted, however, that MemTracker can only automatically handle state checks and updates that can be expressed as a state transition table, and other checks and state updates would still require software intervention. However, many useful memory checkers can be expressed in terms of state transitions, and we expect that in other checkers MemTracker's state machine can be used as a sophisticated filter to minimize the number of software interventions.

3.7 Summary

This chapter describes MemTracker, a new hardware support mechanism that can be set up to perform different kinds of memory access monitoring tasks. MemTracker associates each word of data in memory with a few bits of state, and uses a programmable state transition table to react to different events that can affect this state. The number of state bits per word, the events to react to, and the transition table are all fully programmable by software. The MemTracker state is kept in main memory and cached on the processor chip, and is looked up and updated by the MemTracker hardware. Any state-event pair can be programmed to trigger execution of a software handler, which is used to report a problem or to handle sophisticated checks or recovery. The rich set of states, events, and transitions supported by MemTracker allows bug checks to proceed with minimal performance overheads. To evaluate our MemTracker support, we map three different checkers onto it, as

well as two different implementations of a checker that combines all three. Even for the most demanding combined checker that maintains 8 bits of state for every memory word, we observe performance overheads of around 3% on average and 14.5% worst-case across SPEC and SPLASH-2 applications.

We examine several approaches to caching MemTracker state on-chip, and find that it is possible to implement MemTracker without significant changes to most of the processor pipeline and L1 caches, by adding two in-order stages to the back-end of the processor pipeline and by using a smaller dedicated L1 state cache. In the L2 cache and memory, MemTracker state is stored just like any other data, without any extra support. With its low performance overhead, simple implementation and improved programmability, we see that MemTracker is well in line with our thesis statement, showing that hardware mechanisms for correctness debugging can be designed to have low cost and low performance overheads.

CHAPTER IV

FLEXITAINT: TAINT PROPAGATION ACCELERATOR

4.1 Motivation

Tainting is a popular technique to track the flow of data values through the program. Since users demand different taint propagation policies depending on their needs, a programmable hardware capable of implementing these different policies would be a valuable addition to the user community. This chapter introduces FlexiTaint, a programmable accelerator for taint propagation. Instead of directly implementing a specific set of tainting policies, FlexiTaint is programmed at runtime to efficiently follow a desired tainting policy. This allows a FlexiTaint-equipped system to implement radically different taint propagation policies for different applications, to upgrade its taint propagation policies as new attacks are devised to circumvent existing policies, and even to use tainting for uses other than attack detection/avoidance. As an accelerator, FlexiTaint is not a comprehensive security solution by itself, but rather a proof-of-concept hardware substrate that can speed up an important class of security solutions, namely taint propagation.

One of the main advantages of taking the programmability approach for taint propagation is reduced risk of obsolescence. Whereas a software tool can quickly be upgraded to guard against new attacks, hardware based tools can only be upgraded by replacing the processor or the entire system. This problem is exacerbated by the need to maintain backward compatibility with existing software – once a hardware mechanism is implemented in a processor, new processors must continue to support that functionality. As a result, a hardware scheme can continue to increase the cost and complexity of systems for years, long after attackers have discovered how to circumvent it. This concern makes it very risky to directly implement any specific taint propagation policy or set of policies in a commodity

processor. Our programmable accelerator approach allows taint propagation policies to be changed by software as new attacks are devised, reducing the risk of hardware becoming obsolete.

The rest of this chapter is organized as follows: Section 4.2 presents an overview of our FlexiTaint mechanism, Section 4.3 presents some hardware implementation details, Section 4.4 presents the setup for our experimental evaluation, Section 4.5 presents our experimental results, Section 4.6 discusses related work, and Section 4.7 summarizes our findings.

4.2 Overview

In this section, we first describe the taint storage mechanism adopted by FlexiTaint. We then present how FlexiTaint performs taint propagation in hardware along with optimizations that minimize the performance impact caused by propagation of taint information.

4.2.1 Storing Taint Information

Taint information must be associated with every word in memory. Previously proposed hardware support for taint propagation stores the taint along with the corresponding data, effectively widening the memory word [18]. This approach has a number of drawbacks. First, it requires non-standard memory modules to keep the extra taint bits with each word. Second, these taint bits are wasted when no tainting is needed. Finally, special hardware and instructions are needed to access these taint bits, which makes bulk-manipulation (e.g., initialization) of taints difficult.

Previous hardware support for tainting also widens each cache block to accommodate the taint bits. Such widening makes the cache larger, more power-hungry, and possibly slower even when tainting is not used. Ho et al. [32] have used page-level tainting to intercept CPU accesses that load data from tainted pages and use emulation to perform taint propagation at byte-level granularity. This reduces hardware complexity but incurs performance slowdown of 26.6$\times$ (average) in emulation mode.

The taint storage approach of FlexiTaint parallels the approach used in MemTracker [76] and HeapMon [62] for their memory state (Chapter 3). In particular, taints for data in an address space are kept as a packed array in a protected area within that address space. Given the address of a memory location, the corresponding taint can be found by simply indexing into this array. This organization allows us to use existing standard memory modules, buses, and caches. Taints can be kept protected from ordinary load/store accesses using existing page access permissions, but the system and library software can temporarily change these permissions to directly access taints for bulk-manipulation (e.g. to initialize them).

The only dedicated storage for taints in FlexiTaint is a separate small (4KBytes in our experiments) L1 cache. This cache provides bandwidth for taint accesses. The alternative would be to store taints in the existing L1 cache and add ports to it. However, this would make the L1 cache significantly larger and slower, affecting its hit latency (even when tainting is not needed).

Seemingly, a drawback of our taint storage approach is that taints now compete with data for space in secondary caches and below. This is similar to state competing with data for space in lower-level caches in MemTracker 3. However, our results (Section 4.5) show that this causes low performance overheads relative to a system with no tainting. If extra area is available, we believe it is better spent increasing the total capacity of the cache (to improve performance with or without tainting), instead of widening each cache block to provide dedicated taint bits.

A final advantage of decoupled taint storage is low-cost support for larger taints. In existing approaches, memory widening must accommodate the largest allowed taint, and those extra bits prove unnecessary if fewer taint bits are actually needed. With FlexiTaint, the packed taint array occupies only as much memory space as needed.

4.2.2 Processing Taint Information

Figure 20 shows how taint processing is integrated into the processor's pipeline in previously proposed schemes where taint information (dotted lines) must flow along with and be processed simultaneously with the data (full lines). Shaded areas in the figure indicate structures that are added or significantly changed to support tainting.

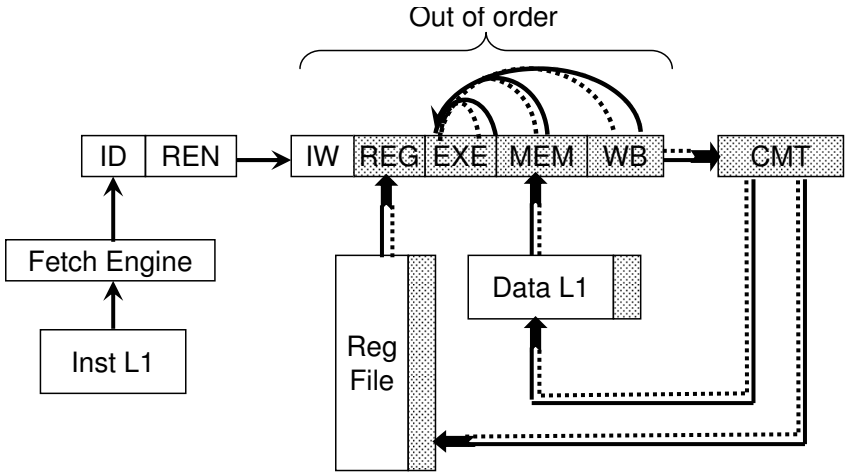


Figure 20: Previous tainting support

Much of the processor's logic and wire complexity involves moving, manipulating, or storing data values. To accommodate the taint along with the data, all these structures must be modified. These ubiquitous changes to the processor core require a tremendous re-design effort and make nearly every part of the core larger with a higher latency. It is unlikely that processor manufacturers will undertake such re-design solely to provide efficient tainting support.

In light of these considerations, we follow a different approach and implement the entire FlexiTaint taint processing accelerator as an in-order addition to the back end of the pipeline, as shown in Figure 21. This strategy has already been used for runtime verification [27], memory checking [76] and speculative memory scheduling [7, 58], and has the advantage of keeping the performance-critical out-of-order dataflow engine largely unmodified. The added taint processing engine is easily turned off and bypassed when it is

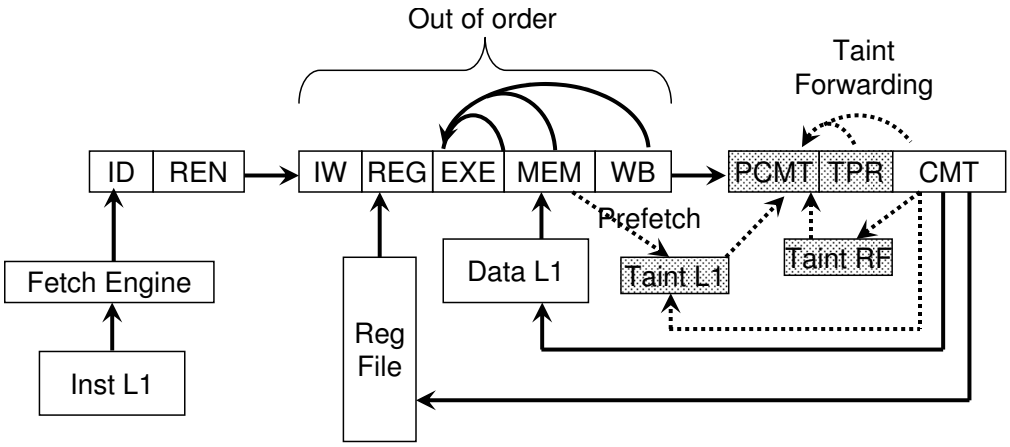


Figure 21: Processor pipeline with FlexiTaint

not needed.

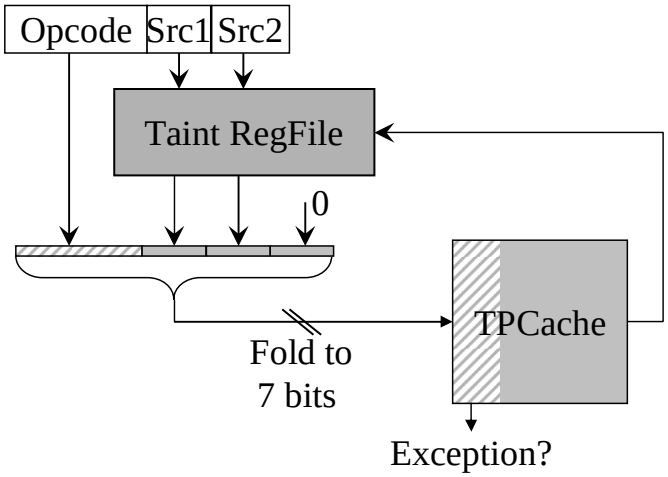


Figure 22: Taint propagation in FlexiTaint

We change the commit stage of the processor to pass instructions on to the FlexiTaint engine. We call this additional stage pre-commit. The first FlexiTaint stage reads the taints of the register operands from the Taint Register File (TRF) (Figure 22). For load and store instructions, the taint of the memory operand is loaded from the TL1 cache. The next stage looks up the Filter Taint Propagation Table (Filter TPT) to determine if the taint propagation can be done using simple rules. If needed, the taint propagation rule is looked up in a special cache that store such information (Section 4.2.3). Next, the register result taint is written back to the TRF. After a possible TRF update, the instruction is ready to

commit. More details are given in Section 4.3.

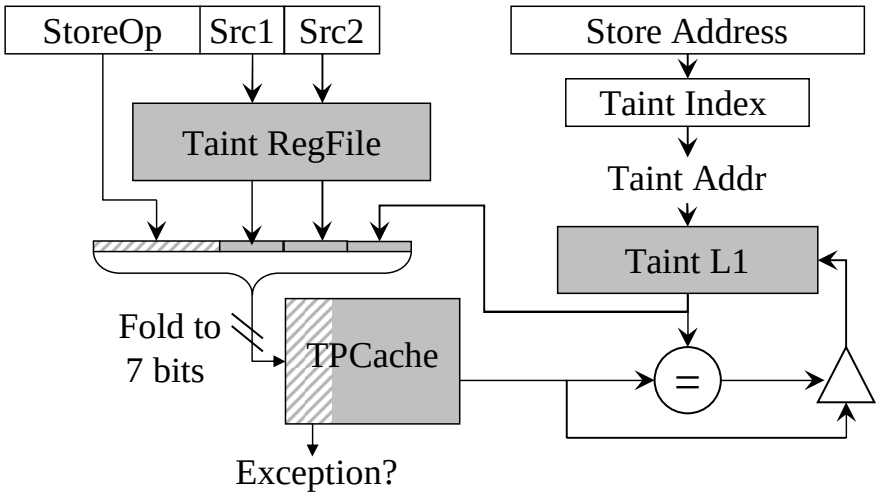


Figure 23: Taint propagation for stores

The commit for store instructions (Figure 23) involves the normal write to the data cache, and a write to the taint cache to update the taint of the destination memory location.

There are several advantages and efficiencies with this new approach to hardware taint propagation. First, the short in-order taint processing pipeline greatly simplifies the taint forwarding logic for both register and memory taints. Second, there is no register renaming, so the TRF only needs to store the taints for architectural registers. Third, support for multiple taint bits only affects the taint processing engine, where the TRF and the forwarding logic are much simpler and smaller than in the main processor core. Store-load taint forwarding uses the same mechanisms as state forwarding in MemTracker (Chapter 3).

This approach also has several possible disadvantages. The main disadvantage is that the latency of misses in the TL1 cache is fully exposed because they stall the in-order taint processing pipeline. Fortunately, taint addresses can easily be computed from data addresses, so we issue a taint prefetch as soon as the data address is available. Because the taint access pattern is a more compact version of the data access pattern, a TL1 prefetch miss is often overlapped and its latency hidden by a DL1 miss. As a result, when a load instruction comes to the pre-commit stage and requests its taint from the TL1 cache, the taint prefetch is usually already complete and the access is a TL1 hit.

The second disadvantage of our back-end taint processing approach is that it could delay instruction commit by several cycles, increasing the pressure on the ROB, physical registers, and other processor resources. However, with the use of structures like TPC and a Filter TPT, we find that this delay is short and in our experiments it has a modest effect on performance.

A third disadvantage of our scheme is that the decoupled approach to data and taint storage can result in consistency problems in multiprocessors. More specifically, a data write and its corresponding taint write must happen atomically to prevent a read from getting the new data with the old taint or vice versa. Similarly, a data read and a taint read must also happen atomically. We found that read atomicity can be provided by leveraging existing load-load replay mechanisms. We also find that most taint writes are silent writes [41] that can easily be eliminated. For non-silent taint writes, we ensure atomicity by making sure that both writes (taint and data) are L1 cache hits before allowing either of them to modify its cache block.

Finally, a fourth potential issue in our FlexiTaint engine is that dependences between instructions may cause in-order stalls and create a performance bottleneck. Fortunately, our FlexiTaint engine has two major advantages over ordinary in-order processors. First, in-order processors suffer stalls when they encounter an instruction that depends on a long-latency instruction. In FlexiTaint, all instructions have similar short-latency taint lookups, and cache misses are largely eliminated by prefetching. The second advantage is that most taint propagation operations are simply copying input taints to the destination, which allows us to eliminate most dependences between same-cycle taint propagation operations (see Section 4.2.5).

4.2.3 Programmable Taint Propagation

Because it is an accelerator that should be able to implement many different tainting policies, FlexiTaint allows software to compute the resulting taint for a given combination of

the instruction opcode and input taints. However, software intervention for every instruction would introduce huge overheads. To avoid these overheads, FlexiTaint uses a Taint Propagation Cache (TPC) to memoize resulting taints for recently seen combinations of opcode and input taints.

The TPC is indexed by concatenating the opcode (or opcode class) and the taints of all source operands (Figure 22). Each entry in the TPC contains the resulting taint, and also a bit that indicates whether an exception should be raised when this combination of opcode and input taints is encountered. If the TPC lookup results in a TPC miss, a software handler is invoked to compute the resulting taint for that combination of opcode and input taints. This is then inserted into the TPC similar to how TLB entries are filled by software TLB miss handlers. This approach allows us to specify a taint propagation policy simply by writing a TPC miss handler. The address of this handler is kept in a special CPU control register (TPC Handler Register).

In our experiments, we use a small (128-entry) on-chip TPC, which is directly mapped to allow single-cycle lookups. To avoid caching stale TPC entries, the TPC is flash-cleared whenever the TPC Handler Register is changed. This allows us to change the taint propagation policy (e.g. on a context switch) by just writing a new handler's address into the TPC Handler Register. Note that tainting is often used to detect attacks, so attackers must not be allowed to change the TPC Handler Register or the code of the handler itself. For this reason, the TPC Handler Register can only be modified in kernel mode.

4.2.4 Taint Manipulation Instructions

In most taint propagation schemes, there are high-level events that affect taint propagation but are not readily recognizable at the hardware level. For example, an input tainting scheme typically untaints data that has been range-checked to avoids false alarms for jump-table implementations. However, in most ISAs a range-check involves a sequence of instructions that are difficult to recognize by the hardware as a range-check.

To allow high-level events to be conveyed to our FlexiTaint hardware, we add a small number of new `taintr` and `taintm` opcodes. These instructions are treated as no-ops in the main processor pipeline, but are processed in the FlexiTaint engine. A `taintr` instruction has a register-to-register format, and based on the opcode and the the input taint values, FlexiTaint performs a TPC lookup to determine the new taint of the destination register. This instruction allows us to change the taint associated with a register value, without changing the value itself. Similarly, a `taintm` instruction has the format of a store operation, but only affects the taint of the memory location. The system software can use these `taintr` and `taintm` instructions to mark high-level events and set the propagation rules for these instructions to achieve the needed taint propagation actions. For example, a `taintm` instruction can be added to the *message receive* code to taint the input data, and `taintr` can be used to, for example, untaint a value that has been range-checked. Note that the mapping between high-level events and these new opcodes is determined by the programmer or the system, not by the ISA or hardware itself. To indicate a specific high-level event, we choose an unused opcode, put it in the code to signal the event, and change the TPC miss handler to perform correct tainting for the event when that opcode is encountered.

Instead of using new opcodes, high-level events could be indicated by trapping into the system, which can then directly read/write the taint array in memory to implement the needed tainting behavior. However, such traps may add significant overheads when high-level of interest are frequent.

Finally, we note that any hardware taint propagation scheme requires changes to the system and libraries to indicate high-level events that cannot be identified by hardware alone. Our implementation of FlexiTaint differs only in that it provides a generic set of ISA extensions for this purpose. This is consistent with FlexiTaint's role as an accelerator for taint propagation - it speeds up the time-consuming activity of instruction-to-instruction taint propagation, and relies on existing taint propagation schemes for system-level support

and for specific sets of rules it is programmed with.

4.2.5 Fast Common-Case Taint Propagation

Although the TPC is small, if it is accessed for every instruction it would need to be multi-ported to keep up with the throughput of the multiple-issue processor core. However, multi-porting could slow the TPC down, make it power-hungry, and make future enhancements or upgrades to larger TPCs costly.

To reduce the number of TPC accesses, we rely on two key observations that hold for most (but not all) types of instructions in the schemes we studied. First, if inputs are untainted (zero-taint), the output is also untainted. Second, if only one of the operands has a non-zero taint, the result taint is simply a copy of the non-zero input taint. This filters most of the TPC lookups and enables the use of single-ported TPC.

To exploit these observations, we use a programmable *Filter Taint Propagation Table* (Filter TPT) to selectively enable these optimizations for opcodes to which they are applicable according to the current set of taint propagation rules. The Filter TPT is indexed only by opcode, and each entry has only two bits that tell us which common-case optimizations can be used (Table 3). With 256 opcodes, the Filter TPT is a simple 512-bit table (no tag checks). Although it has multiple read ports to support separate lookups for each instruction issued in a cycle, this small table uses little on-chip area and is fast enough for single-cycle lookups.

We note that these optimizations are based on observations that were made in our two example taint propagation schemes (Section 4.4). It is possible to devise a set of taint propagation rules for which TPC lookups are always needed (all Filter TPT entries are **00**). Fortunately, now and in the foreseeable future the most common use of taint propagation is tainting of input-derived data to detect security violations. These schemes are similar to the first example tainting schemes used in our experiments, so we expect such schemes to show similar benefit from the Filter TPT.

Table 3: Meaning of Filter TPT entries

Value	Meaning
00	Use Taint Propagation Cache (TPC) for this opcode.
01	If all source taints are zeros, destination taint is zero. Otherwise, like **00** (use TPC).
10	If only one non-zero source taint, copy it to destination taint. Otherwise, like **01**.

Another benefit of using our Filter TPT is that its optimizations also allow us to eliminate most of the dependences between same-cycle instructions. If the TPC lookup is not needed, note that the resulting taint is either zero or equal to the taint of one of the operands. In such cases, the "computation" of the taint is trivial because the taint propagation is simply "do nothing" or "copy input taint to output". Also, back-end pipeline stages are in-order, hence, the forwarding logic required for taint propagation is far simpler that the one required for data. A similar idea (but with a different implementation) was used for elimination of move instructions in RENO [54]. In FlexiTaint, we have the added advantage that most of the taint propagation operations are taint moves, even when the corresponding data operation for the same instruction is not a move. In cases when such optimizations cannot be used and taint forwarding is needed between same-cycle dependent instructions, such dependent instructions are delayed until the next cycle. This may result in delaying retirement of later instructions and an increasing the execution time, but our experiments show such delays to be rare (Section 4.5). So, we choose to simplify FlexiTaint hardware.

4.3 Implementation

With FlexiTaint, the front-end and the out-of-order dataflow engine of the processor core are largely unmodified. The most significant modification to these parts of the processor is that load and store instructions also compute the taint address and issue a non-binding taint

prefetch into the TL1.

The main FlexiTaint pipeline (Figure 21) begins when the instruction is otherwise ready to commit. As a result, FlexiTaint receives instructions in-order, does not receive any wrong-path or otherwise speculative instructions, and gets already-decoded instructions. This greatly simplifies the implementation of our FlexiTaint engine.

FlexiTaint starts off by fetching source taints, (in parallel) looking up the Filter TPT, and (also in parallel) checking dependences. The next step is to check which source taints are zero and whether the value found in the Filter TPT allows us to use a common-case optimization. If one of these optimizations can be used, the taint propagation is trivial - the resulting taint is either zero (if all source taints are zero and the Filter TPT entry has a value other than 00) or equal to the non-zero source taint (if there is only one non-zero source taint and the Filter TPT entry is 10). If the Filter TPT and the source taints are such that a TPC lookup is needed, the next step is to look up the TPC entry using the opcode and the source taints as an index and tag. In this case, a same-cycle dependent instruction (and all subsequent instructions) is stalled until the next cycle. If no TPC lookup is needed, same-cycle forwarding of the original source taint (see Section 4.2.5) is used to avoid in-order stalls. Note that multiple back-to-back forwardings may be needed in the same cycle. We limit such forwarding to one per cycle to reduce the complexity of forwarding logic because our experiments do not show any actual instances where multiple dependent instructions in a single cycle need TPC lookups and forwarding.

After the resulting taint is determined, it is written to the Taint Register File (TRF) if the instruction's destination is a register. Note that TRF could need multiple ports in a superscalar architecture. In order reduce cost and power, we keep it single-ported and our experiments do not show any significant slowdown because of using a single port. After the TRF write, the instruction is ready to commit. In our current implementation, the FlexiTaint pipeline has a total of four stages in addition to the regular pipeline, two to look up the Filter TPT, TL1, and register taints, one stage for actual taint propagation (TPC lookup or trivial

propagation with same-cycle forwarding), and one to finally commit.

Another consideration is handling of TPC misses. A TPC miss is an exception that triggers execution of a software handler. We find that such exceptions are very rare in our experiments due to a combination of several factors. First, most dynamic instructions are amenable to common-case optimizations and do not access the TPC at all. Second, for instructions that do access the TPC, there is significant locality in the values of source taints and opcodes. Some opcodes such as `add`, `sub` which frequently have special taint propagation rules in several commonly used policies [50, 68], are used much more frequently than others and some taint values are much more common than others.

4.3.1 Multiprocessor Consistency Issues

Outside the processor and L1 caches, FlexiTaint memory taint values are stored like any other data, so they are automatically kept coherent in a multiprocessor system. However, FlexiTaint does raise a few issues with respect to consistency. In particular, sequential consistency and several other consistency models assume that a load or a store instruction appears to execute atomically.

The main problem is that FlexiTaint stores taints in memory separately from the corresponding data. As a result, a load (which reads the data and its taint) on one processor and a store (which writes both data and its taint) on another processor may be executed such that, for example, the load obtains the new data but the old taint.

To prevent this inconsistency, data and the taint must be read atomically in a load and written atomically in a store. Load (read) atomicity must ensure that the data value is not changed between the time data is read in the main processor core and the time the taint is read in the FlexiTaint pipeline. Existing replay traps can be leveraged to accomplish this by treating data loads as "vulnerable" to invalidation-caused replay until the corresponding taint is read. We implemented this behavior, and in our experiments we observe practically no performance impact due to such replays because they are exceedingly rare.

Write atomicity for a store instruction is more challenging. To prevent speculative writes to the L1 cache, modern processors delay cache writes until the instruction can commit. When FlexiTaint is active, we also delay the taint write until the instruction commits. As a result, both writes (taint and data) need to be performed atomically at commit time. To simplify the implementation, we change the commit logic to only perform both writes if both are hits - if either data or the taint write is a miss, we do not allow the other to modify its cache. To accomplish this, we check the hit status of both accesses after tag checks, and suppress the actual write in one cache if the other indicates a miss. Once the miss is serviced, both writes are re-tried. There are several factors that prevent this from having a significant effect on performance. First, a TL1 access is nearly always a hit (due to prefetches), so DL1 write hits are rarely delayed by TL1 misses. Second, a tag check in the small TL1 are completed well before the DL1 tag check, so the propagation of the TL1 hit signal does not delay a DL1 hit. Finally, many taint writes are silent writes [41]. Because FlexiTaint already reads the memory location's taint for store instructions (Figure 23), we compare the new taint with the old one and only write the new taint if it differs from the old one. This optimization allows most store instructions to write only data, and also reduces the coherence traffic on memory blocks that store taints because they become dirty less often.

4.3.2 Initialization and OS Interaction

With FlexiTaint, the processor context of a process is extended to include the TPC Handler Register (TPCHR), the FlexiTaint Configuration Register (FTCR), the Memory Taint Base Register (MTBR), and the Filter TPT content. The TPCHR contains the address of the TPC miss handler, and is discussed in Section 4.2.3. The FTCR contains the taint size, which can be from zero to sixteen bits in our current implementation. Taint size of zero bits indicates that tainting is not used, and it turns off taint propagation circuitry and the TL1 cache. The MTBR contains the virtual address of the packed array that stores taints of

memory locations. The Filter TPT is already discussed in Section 4.2.5.

To initialize FlexiTaint, we allocate the memory taint array, initialize it, and protect it from ordinary user-level accesses so only FlexiTaint-initiated accesses can modify these taints. Next, we load the address of the TPC miss handler into the TPC Handler Register. We then load the filtering rules into the Filter TPT and the address of the memory taint array into the Memory Taint Base Register. Finally, we write the number of taint bits to the FlexiTaint configuration register, which enables the FlexiTaint mechanism.

For context switching, we simply save/restore the three registers (TPC Handler, Memory Taint Base, and FlexiTaint Configuration) and the Filter TPT content to/from the context of the process. This save/restore is very fast: only three additional registers are saved/restored, and the Filter TPT is very small. In our implementation, it is only 64 bytes (512 bits) in size.

As our memory taints have their own virtual and physical addresses, they are also fully compatible with OS mechanisms such as copy-on-write optimizations on process forking, paging and virtual memory, disk swapping, etc.

4.4 Evaluation Setup

To evaluate our FlexiTaint accelerator, we use two example taint propagation schemes. The first is input tainting similar to dynamic information flow tracking [68]. This scheme is intended to be representative of tainting schemes that look for security violations by tainting input-derived data and detecting when such data is used in insecure ways (e.g. as data pointers, or jump addresses). The second example scheme "taints" values that are valid heap pointers. This can be used to speed up pointer identification for memory leak detection. This scheme's propagation rules are very different from those of input-tainting, so it also illustrates how easily FlexiTaint can be used with different taint propagation policies. We note that this evaluation is not intended to show that FlexiTaint can detect a specific attack or a class of attacks. FlexiTaint can be programmed to follow a wide variety of taint

Table 4: External Input Tracking.

Instruction	Input-taint propagation rule
ALU-Op R1,R2,R3 (add, sub, etc.)	Taint(R1) = Taint(R2) OR Taint(R3).
mov R1,R2	Taint(R1)=Taint(R2).
ld R1,offset(R2)	Taint(R1)= Taint(R2) OR Taint(Mem[R2+offset]).
st offset(R1),R2	Taint(Mem[R1+offset])= Taint(R1) OR Taint(R2).
taintm0 offset(R1)	Taint(Mem[R1+offset])= INPUT-TAINT.
Jump R1 (branch, jump, etc.)	If Taint(R1)= INPUT-TAINT, raise exception.

propagation schemes, and the attack detection comes from these schemes. Consequently, we do not claim taint propagation schemes in Tables 4 and 5 as our contributions. They are merely example schemes to show how FlexiTaint can be used and to evaluate its impact on performance.

4.4.1 Taint Propagation Schemes

Table 4 shows the rules we use for input tainting. In this table, `taintm0` is the first of our `taintm` opcodes, which we added to input/output libraries to indicate memory into which external inputs were just received in `read` and similar input functions.

These rules can easily be converted into TPC and Filter TPT entries. For example, according to the table, the TPC entry for a *store* instruction with both the address register and value register tainted would be to raise no exception and to taint the target memory location. Also, the same rule allows us to set the Filter TPT entry for `store` to ”10”, avoiding TPC lookups for a `store` with input taint combinations other than the one just described.

Our simulator uses the MIPS instruction set, where R0 is hard-wired to the value of zero and `add rX,rY,R0` is used to move values from `rY` to `rX`. In keeping with this, we also hard-wire the taint for register R0 to zero (no taint), so no separate rule for *mov* is needed. Alternatively, uses of `add` as `mov`, zero-out uses of `xor` and `and`, etc. can be decoded as separate opcodes that have their own taint propagation rules.

Table 5: Heap Pointer Tracking.

Instruction	Pointer-taint propagation rule
add R1,R2,R3	Taint(R1)=Taint(R2) OR Taint(R3). Exception if Taint(R1) AND Taint(R2)
sub R1,R2,R3	Taint(R1)=Taint(R2) XOR Taint(R3).
mov R1,R2	Taint(R1)=Taint(R2).
ld R1,offs(R2)	Taint(R1)= Taint(Mem[R2+offs]).
st offs(R1),R2	Taint(Mem[R1+offs])= Taint(R2).
taintr0 R1	Taint(R1)=POINTER-TAINT.

For heap pointer tracking, we use rules shown in Table 5. The `taintr0` is the first of our `taintr` opcode, which we added to memory allocation libraries to indicate return values of heap allocation functions such as `malloc`.

Note that these propagation rules are different from those in Table 4. For example, the `store` instruction only propagates the pointer-taint of the value register, but ignores the heap-pointerness of the address register. As a result, we can only set the Filter TPT entry for `store` to "01" (see Table 3).

For propagating both taints, we use a two-bit taint where the first bit is propagated according to Table 4 (input tainting) and second bit is propagated according to Table 5 (pointer tainting). For example, the TPC entry for a `store` instruction with the address register taint of "01" (heap pointer) and value register taint of "10" (input-derived value), the exception bit would not be set and the taint for the result (memory location) would be "10" because the input taint of the value register is propagated but the pointer taint of the address register is not. Note that the Filter TPT entry for `store` can only be set to "01" to eliminate TPC lookups when both source registers are untainted.

4.4.2 Benchmark Applications

We use all applications from the SPEC CPU 2000 [67] benchmark suite. For each application, we use the reference input set in which we fast-forward through the first 10% of the execution to skip initialization, and then simulate the next one billion instructions in

detail. We note that our fast-forwarding must still model all taint creation and propagation to provide correct taint state for the simulation. In order to evaluate the multi-threaded workloads, we simulate benchmarks from the Splash2 [78] suite (no fast-forwarding).

4.4.3 Simulator and Configuration

We use SESC [57], an open-source execution-driven simulator, to simulate an 8-core system with Core2-like, four-issue out-of-order superscalar cores running at 2.93GHz. Data L1 caches are 32KBytes in size, 8-way set-associative, dual-ported, with 64-byte blocks. L1 caches have 2 cycle hit latency and 1 cycle miss penalty. The shared on-chip L2 cache is 4MBytes in size, 16-way set-associative, single-ported, with 64-byte blocks. L2 caches have 10 cycle hit latency and 4 cycle miss penalty. The processor-memory bus is 64 bits wide and operating at 1333 MHz. Round-trip memory latency is 490 cycles. In FlexiTaint configurations, taint L1 caches are 4KBytes in size, 4-way set-associative, dual-ported, and also with 64-byte blocks.

4.5 Evaluation

In this section, we first evaluate our FlexiTaint hardware with different taint propagation policies described in Section 4.4.1. Second, we show the effect of some common-case optimizations that we presented in Section 4.2.5 and silent writes that happen on taint information. Third, we perform sensitivity analyses for different L1 Taint Caches with varying capacity and cache block size. We then compare FlexiTaint with prior mechanisms and finally, conduct validation experiments to verify the correctness of our implementation.

4.5.1 Performance with different taint propagation policies

We conduct experiments to evaluate the performance of FlexiTaint when it is programmed to implement input tainting, pointer tainting, and also when it is programmed to simultaneously implement both schemes. Figure 24 shows the execution time overhead for all SPEC2000 and Splash-2 applications. For SPEC2000, we observe worst-case overheads

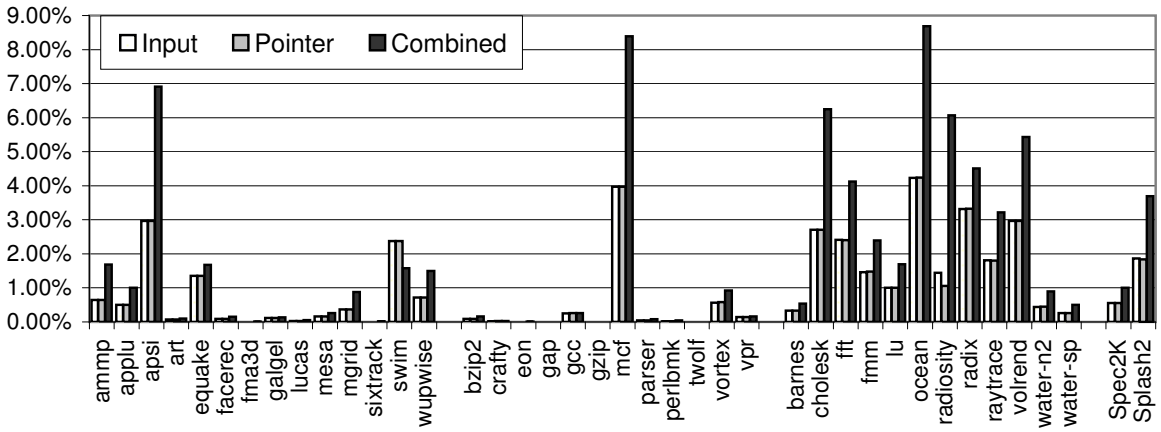


Figure 24: Performance overhead of taint propagation with FlexiTaint.

of 8.4% (in mcf) and average overheads of about 1%, even for the combined taint propagation scheme. In benchmarks with above-average performance overheads, most of the overhead is caused by increased L2 miss rates and increased L2 port contention because the L2 cache is used for both data and taints. The L2 capacity problem is also dominant in most applications, which explains the small differences in overheads between one-bit and two-bit schemes. In swim, the overhead in the combined two-bit scheme is slightly lower than for one-bit schemes. As different memory locations are accessed for taints when taint sizes are different, the overlap of cache misses with other operations in these executions is also different. In swim, the combined scheme suffers more cache misses (as expected), but in single-bit schemes there is less overlap and, as a result, they suffer more overhead.

For Splash-2 benchmarks, we observe modest performance overheads - about 3.7% on average and 8.7% worst-case (in ocean). Both input and pointer taint propagation schemes show similar overheads for all benchmarks except radiosity where the input scheme has slightly higher overhead than the pointer scheme. This is due to increased number of taints propagated between memory and registers in the input tainting compared to the pointer scheme.

4.5.2 Effect of common-case optimizations

Figure 25 shows a breakdown of dynamic instructions according to which Filter TPT optimizations are applied (data shown is for the 2-bit combined tainting scheme). Dynamic instructions that could not benefit from Filter TPT common-case optimizations are further classified into those that hit and those that miss in the TPC. Note that the instructions shown as "No Taint" are not all the instructions whose operands carry no taint. Instead, these are instructions whose operands carry no taint *and* the Filter TPT allows that opcode to use the "no taint yields no taint" common-case optimization without accessing the TPC. Similarly, "One Source Taint" instructions in Figure 25 are those that have one operand tainted with a non-zero taint *and* the Filter TPT allows the instruction's opcode to use the common-case optimization of copying the non-zero source taint to the destination taint without using the TPC. We find that most of the instructions processed by the FlexiTaint engine can be handled by one of these common-case rules. Instructions that access the Taint Propagation Cache (TPC) are infrequent enough, and represent only up to 1.9% (in *mcf*) of all dynamic instructions. As expected, TPC misses are extremely rare.

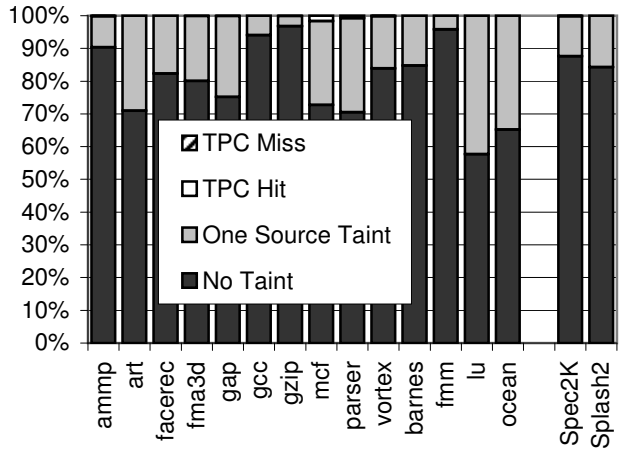


Figure 25: Use of FlexiTaint optimizations.

These results indicate that our combination of TPC handlers for programmability, TPC for memoization of frequently used rules, and Filter TPT for common-case optimizations

can achieve a very high level of programmability with low performance overheads. Also, the TPC can indeed be small and does not need to be multi-ported because most instructions do not actually access it. However, in all of the applications the TPC *is* accessed a non-trivial number of times, which indicates that the TPC and its software miss handler are still needed to support less common taint propagation cases that are specific to each set of taint propagation rules.

4.5.3 Effect of silent taint writes

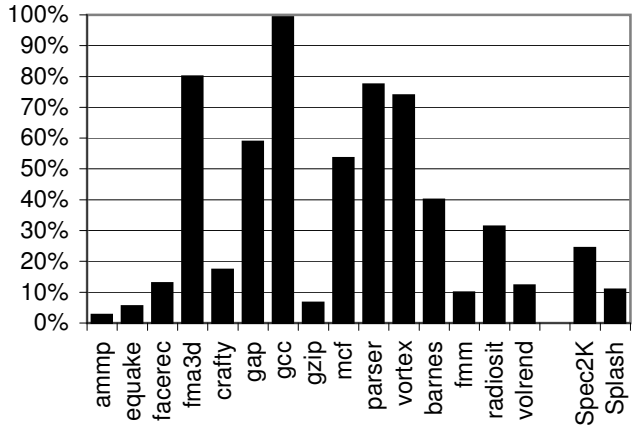


Figure 26: Non-silent taint writes.

Figure 26 shows the number of non-silent memory taint writes as a fraction of the number of dynamic instances of store instructions processed by the FlexiTaint engine. We only show a subset of applications to illustrate the range of the fraction. The averages for SPEC2000 and Splash-2 shown in the figure cover all of the applications. Because every store instruction produces a resulting taint, without detection of silent taint writes, all store instructions would cause FlexiTaint to write the resulting taint to the TL1 cache. From this figure, we observe that in many benchmarks nearly 80% of store instructions do not change the taint of the target memory location. On the other hand, in benchmarks like gcc nearly 99% of stores have non-silent taint writes.

4.5.4 Sensitivity Analysis

The TL1 cache is the largest on-chip structure we added to support FlexiTaint, and we used 4KByte TL1 caches in our experiments. However, we performed additional experiments with 2KByte and 8KByte TL1 caches to determine how the size of the TL1 affect the performance of FlexiTaint. These experiments indicate that a larger 8KB TL1 results in no noticeable performance improvement over our default 4KB TL1 cache. For the smaller 2KByte TL1 cache, we find that the performance overhead increases to 2.8% on average and nearly 14% worst-case (in *mcf*). We conclude that our default 4KByte cache is well-chosen for one- and two-bit taints, but a scheme that uses four-bit taints may need a larger (e.g. 8KByte) cache to avoid some increase in overheads due to TL1 contention.

Our results in Figure 24 show that multi-threaded (Splash-2) applications have higher overheads than single-threaded (SPEC) ones. This is largely due to false sharing of taint blocks. A single taint block corresponds to numerous data blocks: with 2-bit taints, a single taint block corresponds to 16 data blocks. When two processors access different data blocks, the taints for those data blocks can be in the same taint block, causing false sharing.

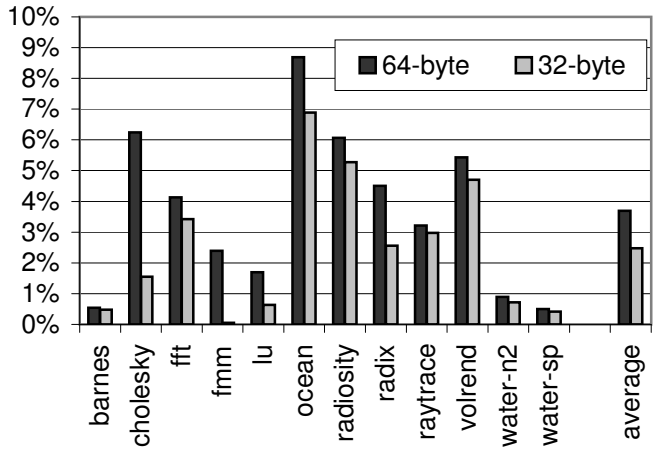


Figure 27: Effect of TL1 line size in Splash-2.

There are several well-known ways to reduce false sharing. One would be to pad and

align data structures. Several Splash-2 applications already use padding to reduce false sharing on data blocks, but the granularity of this padding is insufficient for taint blocks. If the L2 cache is shared, false sharing of taints can be reduced using a smaller block size in the L1 taint cache. Figure 27 compares performance of FlexiTaint with 64-byte and 32-byte blocks in the taint cache (DL1 and L2 use 64-byte blocks in both configurations. We observe performance improvements with smaller taint block sizes, and in several applications this improvement is dramatic. This confirms that false sharing is indeed present, and also shows that it can be reduced without affecting the design of existing DL1 and L2 caches.

With private L2 caches, coherence actions occur between L2 caches. Even if the block size in the TL1 cache is smaller, a TL1 miss still results in a coherence request for an entire larger L2 block. A possible solution would be to use a sectored L2 cache. A data L1 miss would then fetch an entire block into the L2 cache, but a taint L1 miss would only fetch a particular sub-block (sector).

4.5.5 Effect of limited programmability

To demonstrate the benefits of FlexiTaint's full programmability, we artificially limit its programmability to only handle propagation rules that can also be handled by previously proposed hardware support, then use the resulting accelerator on the heap-pointer tracking scheme. Most of the rules for heap-pointer tracking (Table 5) can be handled by this limited scheme. However, rules for `add` and `sub` cannot. In all input tainting schemes, both of these opcodes simply perform a logical OR of input taints to produce the propagate the output taint, without raising any exceptions. In heap-pointer tracking, addition or subtraction of two heap pointers produces a non-pointer, and addition of two pointers is meaningless and indicates a probable bug. As a result, the heap-pointer tracking rule for `add` is to propagate the taint if only one source operand is pointer-tainted, but to raise an exception (invoking a software handler to record a possible bug) when both operands are pointer-tainted. The heap-pointer tracking rule for `sub` is to propagate the taint if only

one source operand is tainted, but produce an untainted result if both source operands are tainted. Effectively, the taint propagation rule for `sub` is a logical XOR of input taints. A limited-programmability scheme must raise an exception for each `add` and `sub` instruction and implement correct taint propagation for these instructions in an exception handler. In our evaluation, we do not actually implement this exception handler. Instead, we model its overhead by adding a (rather optimistic) 5-cycle penalty each time an `add` or `sub` is encountered.

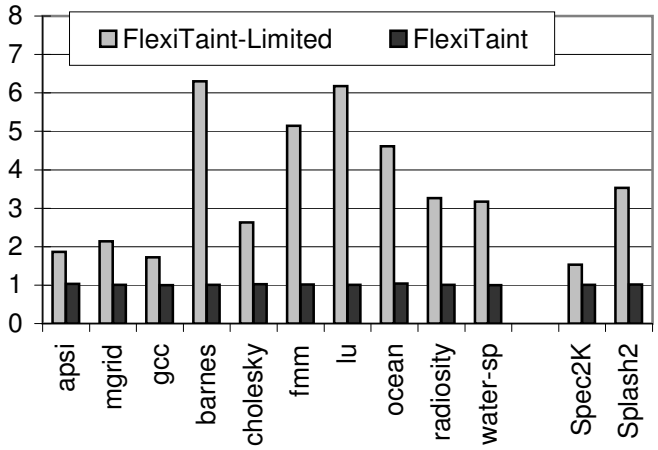


Figure 28: Effect of limited programmability.

The result of this evaluation is shown in Figure 28, which shows execution times with a limited-programmability scheme and with our full FlexiTaint mechanism. Execution times are normalized to a baseline that does no taint propagation. These results indicate that limited-programmability hardware support incurs large performance overheads when it is used on schemes that are not sufficiently amenable to the particular programmability limitations.

Note that we do not claim that other hardware schemes cannot be extended to support the particular taint propagation rules needed to support `add` and `sub` for heap-pointer tracking. For example, the original description of Raksha [18] includes no support for XOR-in input taints, but such support can easily be added.

We argue that the needs of future taint propagation schemes are difficult to predict at

hardware design time, and we claim that by providing more programmability we run far less risk of being unable to efficiently handle those future schemes. In other words, any existing hardware mechanism can be extended with a setting that allows it to support a particular tainting rule it currently does not support, but this obsoletes systems that have the previous iteration of the hardware support. Our approach is to avoid this by designing our FlexiTaint accelerator to be highly programmable to begin with.

4.5.6 Validation

As mentioned previously, input-tainting itself is not our contribution, and we use heap-pointer tracking mostly as an example of a scheme with propagation rules different from those in input tainting. Our FlexiTaint accelerator is intended to provide a programmable, low-cost, and implementable substrate that allows implementation of various taint propagation schemes with low performance overheads. As a result, our evaluation focuses on performance. However, we do verify that our implementation of input tainting and pointer tracking correctly tracks input-derived and pointer values throughout the program. We also validate heap pointer tracking by verifying that tainted variables are indeed heap pointers and that non-pointers remain untainted. We verify input tainting by injecting buffer overflows and verifying that they are indeed detected. Interestingly, some benchmarks (twolf and gcc) have a number of dynamic instructions that produce values tainted with both the pointer taint and the input taint. For such occurrences, we examine the source code of the application to confirm the correct behavior of our scheme.

4.6 Related Work

Static taint analysis was proposed to find format string vulnerabilities in C programs [61] or to identify potentially sensitive data [5]. Taint propagation is also similar to runtime type checking, where each object is "tainted" with its type and operations are checked for type-safe behavior in languages such as Java or CCured [48].

Perl [53] taints external data, and its taint propagation is compiled into the code by the

just-in-time compiler or performed by the interpreter. Newsome et al. [50] use runtime binary rewriting to taint external inputs and propagate taints. Xu et al. [79] *tag* each byte of data, with elaborate policies to track these tags for security.

Hardware support has been proposed to improve performance of tainting and to accommodate self-modifying code and multithreading. Suh et al. [68] propose a low-overhead architectural mechanism that protects programs by tainting data from untrusted I/O and then propagating this taint. It provides some flexibility for particular taint propagation rules. Chen et al. [10] use the notion of pointer taintedness to raise alarms whenever a tainted pointer is dereferenced by the program. Minos [17] extends the microarchitecture with integrity bits and propagation logic that prevents control flow hijacking. TaintBochs [12] taints sensitive data and propagates this taint across system, language, and application boundaries. TaintBochs provides limited configurability to support different tradeoffs between security and the number of false alarms. The RIFLE architecture [72] supports runtime information flow tracking, and allows to enforce their own policies on their programs.

Ho et al. [32] explore hardware support for taint based protection by switching to emulation mode when tainted data is being processed by the CPU. Their scheme is targeted at detecting software attacks that are based on injecting malicious code into the target from the external network. In order to detect such attacks, there are two levels of taint - every byte that is received from the external network is tainted and a page-level tainting bit is set to detect when CPU is reading data from tainted pages into registers.

To our knowledge, Raksha [18] is the most configurable hardware taint propagation mechanism proposed to date. It supports multi-bit taints and has taint propagation registers that can be programmed to implement up to four different policies. Raksha also provides some flexibility in how the taints are propagated for each type of instructions. However, this flexibility is limited mainly to selecting whether or not a given input operand's taint should or shouldn't be propagated to the result, and to selecting whether the taint operands should

be OR-ed or AND-ed. This is sufficient to efficiently implement most variants of existing tainting policies. In contrast, FlexiTaint can be used with *any* set of propagation rules in which the taint of the result depends only on the opcode and the taints of the operands.

Another important consideration for taint propagation schemes is how taints are stored and manipulated. Existing hardware tainting mechanisms tightly couple the data value and its taint: memory locations are extended with extra bits for the taint, and buses and caches are similarly affected. Unlike prior hardware support for taint propagation, FlexiTaint stores and processes taints separately from data. For storage of memory taints, FlexiTaint uses the approach used to store memory state (tags) in MemTracker [76]. Taints are stored as a packed array in virtual memory, allowing use of standard memory modules and existing OS memory management mechanisms. Taint processing in FlexiTaint is implemented as an in-order addition to the back-end of the processor pipeline to minimize impact on the already complex out-of-order core.

4.7 Summary

This chapter proposes and evaluates FlexiTaint, a programmable hardware accelerator for taint propagation. FlexiTaint is implemented without modifying the system bus or main memory, and without extensive modifications to the performance-critical front-end pipeline of the processor or to its out-of-order dataflow engine. FlexiTaint stores memory taints as a packed array in virtual memory, and its taint processing engine is implemented as an in-order addition to the back end of the CPU pipeline. FlexiTaint is also fully programmable, using a Taint Propagation Cache (TPC) backed by a software miss handler to determine the taint of an instruction's resulting data value, using as inputs the type of operation (its opcode) and the taints of the operands. A Filter TPT is also used to reduce the number of accesses to the TPC for common-case optimizations.

Our results indicate that FlexiTaint incurs very low performance overheads, even when using a two-bit taint that simultanously tracks two very different properties with different

taint propagation rules. We also evaluate FlexiTaint on Splash-2 benchmarks and demonstrate that it operates correctly and with low overheads even in a multi-core system. With its low performance overhead, simple implementation and improved programmability, we see that FlexiTaint supports our thesis statement that architects should build hardware mechanisms to implement taint propagation mechanisms that address security and debugging challenges faced by programmers.

CHAPTER V

CACHEDOC: COMPREHENSIVE CACHE MISS CLASSIFIER

5.1 Motivation

Modern processors have several levels of caches to facilitate speedy access to data that are accessed multiple times within short time intervals (temporal locality) as well as data that are accessed within relatively close regions (spatial locality). Memory accesses that do not have data in caches (also known as cache misses) lead to poor application performance and it is necessary to understand their behaviors to minimize their effects.

The infrastructure most commonly used by application writers to diagnose performance issues, for both serial and parallel applications, are performance counters. These are hardware counters that enable collecting statistics from a production environment using real inputs. Today's processors support counting a large number of events with performance counters, making them a powerful and general tool [36]. Hardware support for more sophisticated performance counters [59], attribution of events to instructions [19], and on-the-fly processing of profiling data [46, 47, 84] are active research topics. However, performance counters and other profiling infrastructure currently only monitor hardware events that are relatively simple to detect and classify (e.g., branch mispredictions, retired instructions, cache misses, etc.). While this basic information is sufficient to diagnose some performance limiters such as the saturation of external memory bandwidth, it is not sufficient to diagnose limiters resulting from more complex behavior (e.g., it does not point to a particular type of cache miss in a specific region of code as a performance limiter).

In this chapter, we present CacheDoc, a comprehensive Cache Miss Classifier and explore several designs that classify cache misses into specific categories. We start with ideal/off-line cache miss classification techniques and explore several practical designs that

offer a spectrum of cost-accuracy tradeoffs. We present these cost-accuracy trade-offs and explain our results. The rest of the chapter is organized as follows: Section 5.2 describes our mechanism to classify replacement misses into conflict and capacity misses, Section 5.3 describes our mechanism for identifying and classifying coherence misses, Section 5.4 presents results showing CacheDoc performing comprehensive cache miss classification, Section 5.5 discusses related work, and Section 5.6 summarizes our findings.

5.2 Classification of Replacement Misses

In this section, we first describe an ideal (off-line) scheme to identify conflict misses. We then present the disadvantages in implementing the ideal scheme and describe our practical conflict miss detector and its implementation. Our scheme can be used in any cache, regardless of its level in the memory hierarchy and whether it is private or shared.

5.2.1 Ideal Conflict Miss Detector (I-CMD)

Replacement misses fall into two categories: capacity and conflict. A capacity miss occurs when the requested block was replaced from the cache because the cache did not have enough capacity to accommodate incoming cache blocks. Minimizing capacity misses involves a redesign of the algorithms used in an application (e.g., loop blocking) or upgrading to a processor with larger caches.

In contrast, a conflict miss happens when several blocks map into the same set in the cache and replace each other even when there is enough capacity left in the cache. When the number of these blocks exceeds the cache associativity, a block, A, will be evicted even though better candidates for eviction may exist in the cache in other sets. If A is accessed again before those better candidates are replaced, that access is a conflict miss. That is, a fully associative cache of the same capacity would have kept A in the cache and not incurred the miss. Minimizing conflict misses involves rearranging data so that blocks frequently accessed together map to different sets. This can be done through padding or changing the alignment of data structures.

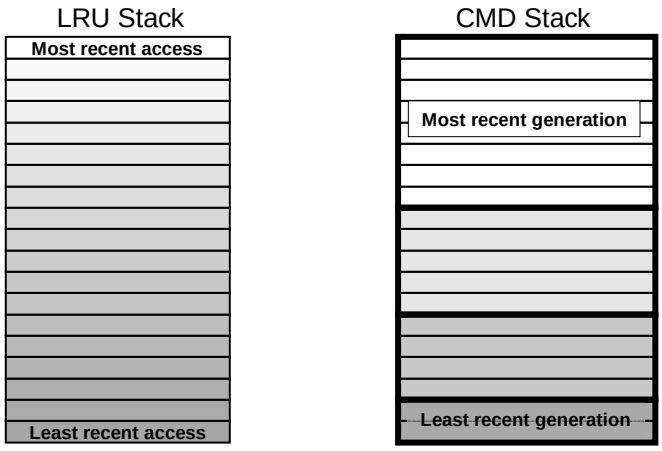


Figure 29: LRU stack (used by I-CMD) and CMD generational stack. The shading shows how recently a block was accessed. In the LRU stack, there is a total ordering among the blocks in the stack. In the CMD stack, no ordering information is maintained within a generation.

For caches that use a least recently used (LRU) replacement policy, the standard off-line technique to detect conflict misses is to maintain an LRU stack with N entries that tracks the blocks that would be present in a fully associative N-block cache. On each access, the block being accessed is placed on the top of the stack. If this block was already present in the stack, it is also deleted from its prior location in the stack. A miss is classified as a conflict miss if, prior to the stack update, the block is found on this N-entry stack. An LRU stack is usually used for off-line analysis of memory accesses [30, 44]. However, implementing this off-line algorithm in hardware for on-the-fly classification would be prohibitively expensive in terms of performance overhead, area cost and power budget.

5.2.2 Practical Conflict Miss Detector

To keep cost low, we propose a new, less complex Conflict Miss Detection scheme that approximates the access-recency information provided by an LRU stack used by I-CMD. In particular, we use a scheme that maintains a number of *generations* that are ordered by age. Each generation consists of a set of blocks, and all blocks in a younger generation have been accessed more recently than any block in an older generation. This means that

the blocks in the youngest generation are the blocks that would be at the top of the LRU stack, the next (older) generation corresponds to the next group on the LRU stack, etc. Figure 29 shows an example using four generations.

Unlike an LRU stack, our generational scheme does not track the recency of access ordering within a generation. For example, if we miss on a block that is found in the youngest generation, and if that youngest generation currently contains 8 blocks, we only know that the requested block is among the 8 most recently accessed blocks. In contrast, an LRU stack would provide precise information—if the block is found in the second position counting from the top of the stack, then that is the second most recently accessed block.

In our scheme, we start with an empty youngest generation and add cache blocks to it as they are accessed until the number of blocks in this generation reaches a threshold number T. After this, we begin a new youngest generation, aging the previous youngest generation and all generations older than it. Because the state used to track generation membership is finite, we only maintain up to K generations. Once we have created the first K generations, the oldest generation is erased and its resources reused to create the new youngest generation. This corresponds to the removal of the bottom entry on the LRU stack when a new entry is added at the top and all other entries are pushed down one place. However, in our generational scheme we remove an entire generation at a time, which results in imprecise classification at the boundary between capacity and conflict misses.

5.2.3 Implementation

Figure 30 shows the hardware components added to a cache in our proposed implementation of CMD and Figure 31 shows how these components are used to detect conflict misses. First, we need to keep track of the current youngest generation since the generations are reused similar to circular buffers. For instance, in Figure 30, the ordering of the generations is 2 (youngest), 1, 0, and 3 (oldest). Second, we maintain an array of counters that tracks the size of each generation. These counters and the youngest generation register contribute

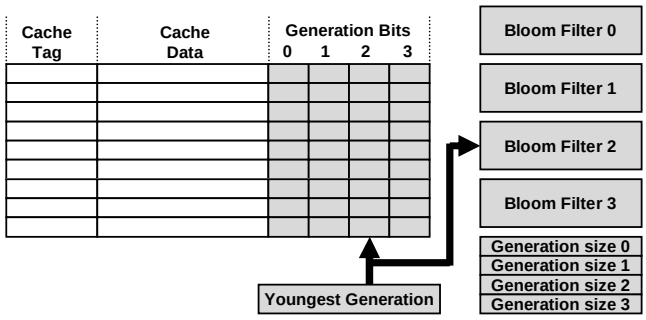


Figure 30: Shaded areas show the hardware added to implement our Conflict Miss Detector (CMD). A 4-generation CMD is shown, at a time when Gen 2 is currently the youngest one.

little to the overall cost of the scheme. Finally, the membership of a generation is split between two structures. Blocks that are present in the cache are tracked using *generation bits*. Blocks that are not present in the cache are tracked using a Bloom filter [4] for each generation.

Generation bits are added to each cache entry. Each bit corresponds to one generation. To put the block into the youngest generation, we simply set the corresponding bit. To determine which generation the block currently belongs to, we determine the first bit that is set (in order of generation age). To clear the state of the oldest generation, we flash-clear all bits that correspond to that generation (and all bits in the corresponding Bloom filter).

When the number of generations is relatively large (8 or more), an in-cache representation that is more compact than generation bits may be preferable. One approach is to record the actual generation number (GN) for each cache block. To put the block into the youngest generation, the current generation number is written into the block's GN. Generation membership of the block can easily be determined by reading its GN. However, to clear a generation we must find all blocks in that generation and change their GNs to a special "no-generation" number. To represent the "no-generation" number, we can add a Generation Valid (GV) bit, which is set only for blocks that belong to some generation. When a new generation is created, we must clear the GV bits for blocks whose generation number matches the new current number. To avoid sweeping through the cache for

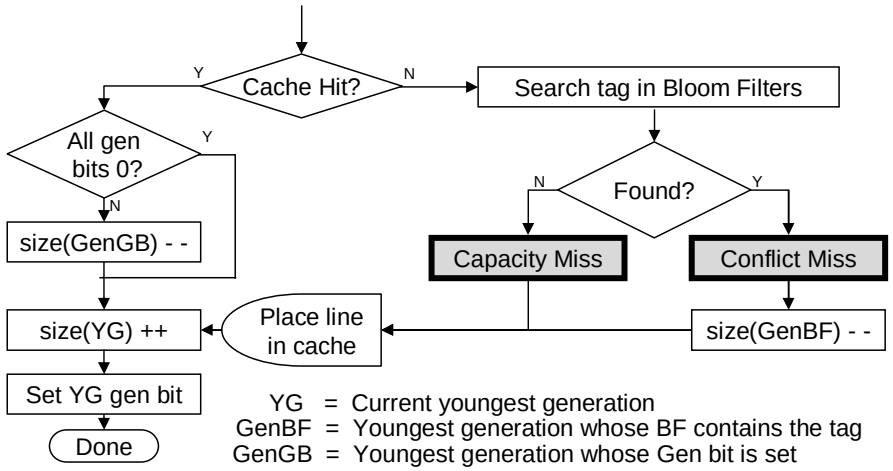


Figure 31: CMD implementation flowchart.

this purpose, we use a sliding window scheme that adds another bit to the GN field itself. When the current generation number changes, if the most significant bit (MSB) changes, all blocks whose GN's MSB matches it have their GV cleared. In other words, when the MSB flips, the oldest generation numbers are changed to "no-generation".

Note that this generation-number scheme uses $2 + log_2(K)$ bits per in-cache block, while the more straightforward generation-bits scheme uses K bits per in-cache block. This means that the generation-bits scheme is preferable when $K \leq 4$.

Each generation uses a Bloom filter to keep track of the blocks that belong to that generation but are not present in the cache. With an N-line cache and K generations, there are K Bloom filters. Each is a three-hash Bloom filter with 4*N/K bits of state, which allows each filter to track all the blocks in its generation with a very low false-positive rate [4]. It is interesting to note that the total size of all the Bloom filters for a given cache is independent of the number of generations (K filters, each has 4*N/K bits). Also, our Bloom filters only need to keep track of *replaced* blocks in a generation, not the entire generation. As a result, we could use considerably fewer than 4N bits and still maintain excellent accuracy [28]. Finally, it should be noted that these Bloom filters are accessed only on cache misses, so they do not affect hit times, can be single-ported, and can be

implemented with cheaper and slower circuitry than the corresponding cache.

Individual entries cannot be deleted from Bloom filters. However, in CMD, an entry in a Bloom filter should ideally be deleted from an older generation's Bloom filter when it is "stolen" by the youngest generation. Instead, we note that older generations are always deleted before younger generations, so it is safe to leave deleted entries in the Bloom filters. The actual generation of a block is now the youngest generation in which that block can be found.

When a cache block is evicted, its generation membership is recorded in the corresponding Bloom filter. When a block is brought into the cache, it will be tracked by the generation bits even if it was already in a Bloom filter. As a result, for in-cache blocks we only need to check and update generation bits, and Bloom filters are only checked on cache misses and updated on cache replacements.

Our CMD scheme should have little effect on the cache hit or miss latency. For a cache hit, the generation bits can be read while the tags are being checked. If the youngest-generation bit is not set already, it needs to be set. This can be accomplished using the same approach used to change the state of the block from clean to dirty on a write. The update of generation size counters, if it is needed, is not on the critical path of the cache access and can be performed in the next cycle.

On a cache miss, the Bloom filter lookup and the CMD classification of the miss can begin as soon as the cache miss is detected, and can easily be completed before the requested data arrives from memory or a lower-level cache.

A final consideration is how to attribute capacity and conflict misses to the instructions that trigger them. Performance debugging in existing processors is typically supported by including several performance counter registers. Each of these registers is typically associated with a control register that determines which event will be counted (e.g., committed instructions, cache accesses, L1 cache misses, etc.). To attribute events to a particular instruction, the counter is typically initialized to some value, and counts down each time

the event occurs. The hardware raises an exception when the count reaches zero, and the program counter value is recorded by a software tool. This sampling approach allows the software tool to provide a programmer an approximation of which instructions triggered the most events of interest, while incurring little execution time overhead. We rely on this conventional mechanism for capacity and conflict miss attribution. However, our scheme can be even more beneficial with more advanced profiling mechanisms such as ProfileMe [19] and stratified sampling [59].

5.2.4 Evaluation

We evaluate our practical generational CMD against off-line I-CMD. We use SESC [57], a cycle-accurate, execution driven simulator. We model 64-core chip multiprocessor. Each core is a 2.93GHz, four-issue, out-of-order processor with a 32KB, 4-way set-associative private L1 cache (2 cycle hit and 1 cycle miss latencies), 512KB, 16-way set-associative private L2 cache (10 cycle hit and 4 cycle miss latencies). All cores share an 8MB, 32-way L3 cache, and the MESI protocol is used to keep L1 caches coherent. The block size is 64 bytes in all caches. Since, conflict misses are more likely in caches with lower associativity [29], we use L1 caches to evaluate effectiveness of our CMD mechanism. New entries are only added to the youngest generation. So the size of a generation will never be more than the threshold T. A seemingly obvious choice is to set T to N/K, where N is the number of blocks in the cache. We conduct two sets of experiments with L1 caches implementing two different cache block replacement policies – Least Recent Used (LRU) and a slightly modified version of Clock-based Not-Recently-Used (NRU) algorithm used in SUN UltraSparc V2 [69]. LRU replacement policy maintains age information for every cache block that tracks the recency of access for that block. LRU chooses the least recently used block (or the block with maximum age) to be replaced with the incoming block. In the Clock-based algorithm, each cache block maintains a ‘used’ bit that gets set when the block is accessed or initially fetched from the memory. In addition, each block also

has ‘allocate’ bit, which gets set when allocated and is cleared when the block is filled with data. Each cache set has a rotating replacement pointer, which is the starting point to find the *way* to replace. On a miss, the algorithm looks for the first cache block that has both ‘used’ and ‘allocate’ bit clear, starting with the *way* pointed by the replacement pointer. If all block have ‘used’ and ‘allocate’ bits clear, all ‘used’ bits are cleared and the scan is repeated. The replacement pointer is then rotated forward one *way*. Both LRU and Clock-based algorithms are used to contrast how well our practical scheme captures conflict misses compared to I-CMD (that uses a fully-associative LRU stack to determine conflict misses).

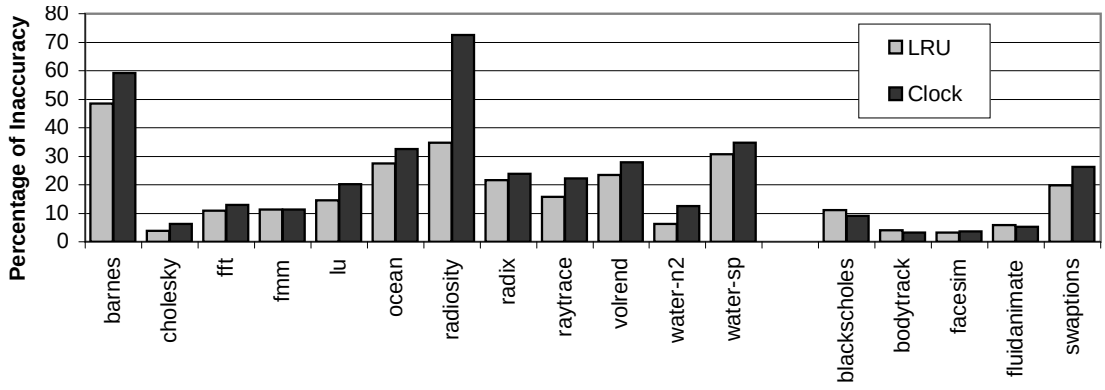


Figure 32: Percentage of inaccuracy for generational CMD on caches with LRU and Clock-based Replacement policies.

In our experiments, we measure the percentage of replacement misses that are not correctly classified by the practical CMD scheme. We call this as *percentage of inaccuracy*. Figure 32 shows the results of our experiments with number of generations, $K = 4$. Each benchmark has two bars which show the percentage of inaccuracy for LRU and Clock-based replacement policies. For LRU replacement policy, all benchmarks except barnes, radiosity and swaptions have inaccuracy less than 20%. On further analysis, we find that in these benchmarks, a significant portion of the conflict misses occur near the bottom of LRU stack in I-CMD scheme. Such misses are incorrectly classified as capacity misses by our

scheme due to loss of information when the older generations are cleared. Even for benchmarks with significant (>30%) inaccuracy, our attribution experiments show that the static instructions that are identified as top ten offenders in I-CMD scheme mostly match with the top ten offenders in the practical scheme. For performance debugging efforts, this information is useful to help programmers fix their code that are hot-spots for conflict misses. Inaccuracy in practical CMD for clock-based replacement policy differs from LRU because of additional "randomness" while choosing blocks for replacement in comparison to LRU. The set of blocks that suffer replacement misses are quite different from LRU. Therefore, the percentage of inaccuracy depends on whether the classification of replacement misses for the blocks chosen by Clock-based policy concur in I-CMD and our practical scheme. In some benchmarks such as barnes and radiosity, there is significantly higher inaccuracy (compared to LRU replacement). In others, the percentage of inaccuracy for both replacement policies differ from each other by <5%. Our attribution experiments show that static instructions that are top offenders in the I-CMD and practical scheme largely match (except for a few cases in barnes and radiosity) and thus, our practical generational CMD scheme helps programmers reach similar conclusions regarding conflict misses.

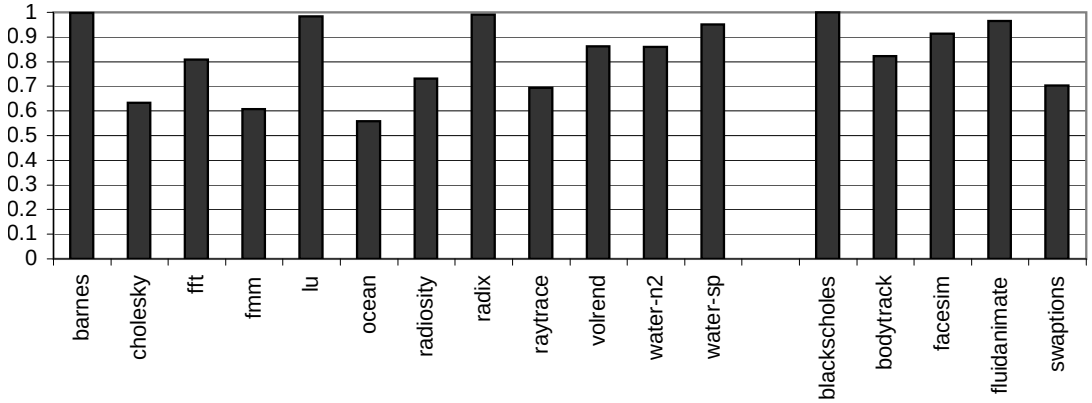


Figure 33: Correlation coefficient between conflict misses suffered by all static instructions in caches with LRU and Clock Replacement policies.

We note that LRU and Clock-based algorithm are two separate replacement policies. As a result, the set of static instructions that suffer replacement misses in a cache with LRU

replacement policy will not be the same as those observed in a cache with Clock-based replacement. By using the LRU stack (I-CMD), we identify all the program counters (static instructions) that suffer conflict misses in both policies and measure the correlation coefficient (also known as Pearson product-moment correlation coefficient [14]) between the corresponding sets of conflicts. This is done to study how closely the Clock-based algorithm approximates the LRU replacement policy in causing conflict misses in caches. A correlation coefficient of +1 means strong correlation and that the conflict-based replacements that happen in both policies are similar or proportional to each other (equivalent modulo scaling). On the other hand, a correlation coefficient of 0 means that they are uncorrelated (independent of each other). Figure 33 shows the result of our experiments. We find that, in certain benchmarks like barnes, lu, radix, water-sp, blackscholes and fluidanimate, we observe a correlation coefficient of 0.95 or above. In other benchmarks, the correlation is not very strong (less than 0.9). From these results, we infer that programmers' intuitive understanding of conflict misses (defined by Hill et al. [30]) is already somewhat blurred by pseudo-LRU replacement policies such as Clock-based algorithm. With this in mind, we believe that our scheme can serve as a good tool for performance debugging purposes even though it is not as precise as I-CMD, because it can still identify static instructions responsible for the majority of conflict misses.

One of the main sources of inaccuracy in our practical CMD scheme stems from the threshold $T = N/K$ being sometimes too small; when an entry being added to the youngest generation is already present in an older generation, it is effectively deleted from that older generation. As a result, older generations become smaller over time due to this "stealing" by the youngest generation. Therefore, the number of blocks in all K generations combined could be considerably less than N. This results in numerous conflict misses (to blocks that would be in the lower part of an LRU stack) being misclassified as capacity misses. We call such conflict misses as "near-capacity" misses because these misses would be capacity misses in a slightly smaller cache. Near-capacity conflict misses by their very nature occur

on blocks that stay in the cache for some time; in contrast, conflict misses on very recently accessed blocks occur after a short time and tend to be a result of thrashing. Thus, near-capacity misses typically have a much smaller impact on performance than other conflict misses.

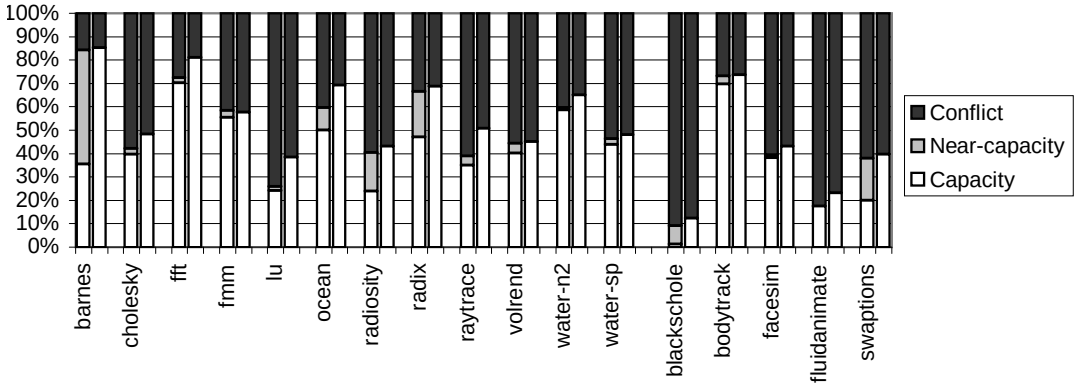


Figure 34: Breakdown of replacement misses. Each benchmark has two bars- the left bar shows the classification according to I-CMD and right bar shows the classification according to our practical CMD.

Figure 34 shows the breakdown of replacement misses. For each benchmark, there are two bars – the first bar shows the breakdown in I-CMD scheme and the second bar shows the breakdown in the practical scheme. Conflict misses in I-CMD are broken down as near-capacity and conflict, that is, misses on blocks that are in the top (most recent) 75% of the LRU stack are still shown as "conflict," but misses on blocks that are in the bottom (least recent) 25% of the LRU stack are shown as "Near-Capacity." We see that our practical generational CMD scheme performs very close to the I-CMD (offline LRU scheme) for more-recent conflict misses, but misclassifies Near-Capacity conflict misses.

Unfortunately, our lack of knowledge about ordering within a generation prevents us from re-balancing generations by moving the oldest block in a younger generation into the next older generation. This means that the only way of controlling generation sizes is the threshold T that determines the final size of the youngest generation. However, if the threshold is chosen to be significantly larger than N/K, insufficient "stealing" by the youngest generation may result in more than N blocks being represented and, consequently,

some capacity misses (to blocks that would be just below the N-th place in the LRU stack) would be classified as conflict misses.

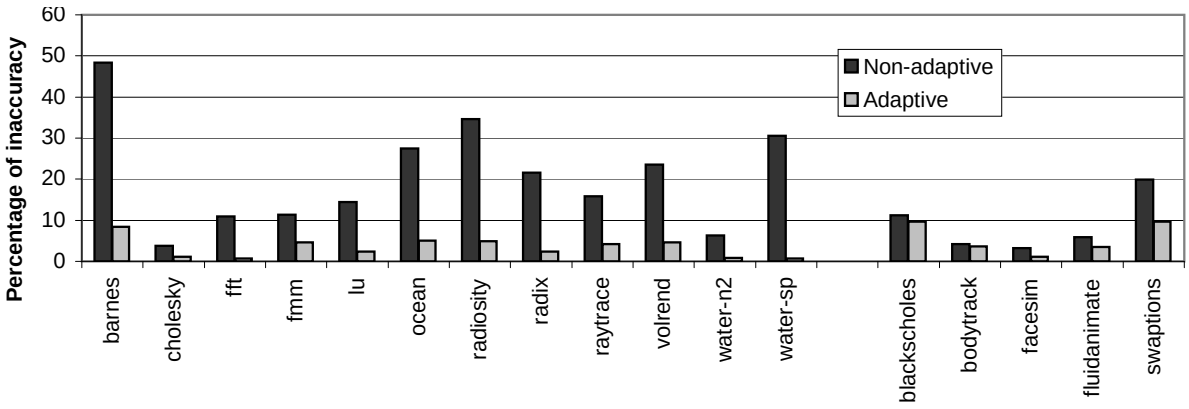


Figure 35: Percentage of inaccuracy in our practical CMD when thresholds in generations are non-adaptive and adaptive.

To manage the trade-off between these misclassifications, there are two possible solutions – i) adaptive threshold, and ii) increasing the number of generations. The adaptive threshold scheme determines the threshold T based on the number of distinct accesses tracked by all generations. Whenever we are about to create a new generation, we determine the total number of blocks represented in all generations. If this number is less than N, the threshold is increased to enlarge future generations. Conversely, if all generations together have more than N blocks, the threshold is lowered to shrink future generations. Finally, we avoid misclassifications of capacity misses by ignoring old generations which are not entirely above the Nth position on an LRU stack. In other words, if the sum of generation sizes for the L youngest generations is larger than N, then generation L and older are ignored because a miss that finds its block in one of those generations may be a capacity miss.

Figure 35 shows the result of our experiments. For each benchmark, the percentage of inaccuracy is studied for both non-adaptive (where T is fixed at N/K) and adaptive schemes. We find that through adaptive threshold, we significantly reduce the percentage of inaccuracy to $<$10% in all benchmarks (maximum of 9.7% in swaptions). The adaptive scheme

is conservative in that it never misclassifies capacity misses as conflict misses; however, it may still misclassify some ”near-capacity” conflict misses to blocks near the end of the (conceptual) LRU stack. Also, near-capacity misses that remain even with adaptive thresholds are less amenable to being avoided through padding and alignment—such fixes may simply convert these misses into true capacity misses. Therefore, we consider misclassification of some near-capacity misses acceptable for performance debugging purposes.

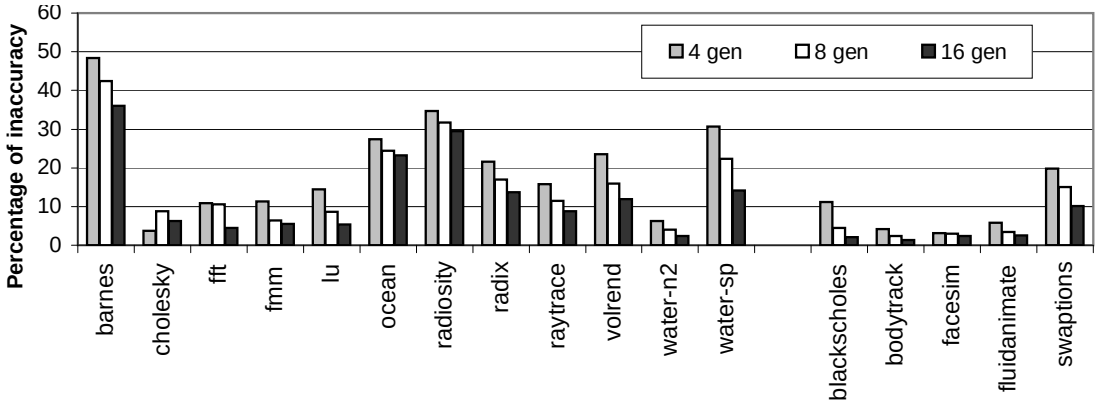


Figure 36: Accuracy of classification for 4, 8 and 16 generations.

A second solution to minimizing inaccuracy is to increase the number of generations used to track conflict misses. When each generation holds N/K entries, with larger K, lesser number of entries (information about certain cache block accesses) are lost when erasing the oldest generation. Figure 36 shows the results of our experiments using 4, 8 and 16 generations. Across all benchmarks, the percentage of inaccuracy drop gradually as we increase the number of generations. However, even with 16 generations, the inaccuracy is still $>$30% in certain benchmarks like barnes and radiosity. This is because, a significant portion of conflict misses occur near the very bottom of LRU stack in the I-CMD scheme which are misclassified in the practical scheme.

Table 6 shows the latency, area, and per-access energy overheads of both the offline I-CMD and our practical CMD schemes. The ideal scheme must update the LRU stack on every access. Since it is essentially a tag array for a fully associative cache, it has an access latency that is more than twice that of the baseline cache and requires up to 3x extra energy

Table 6: Latency, area, and energy overheads.

Private 32KB L1			
	L1 Access Time	**Total L1 Area**	**L1 Power**
Ideal-CMD	18.48%	111.65%	86.46%
Practical CMD	1.48%	3.89%	0.70%
Private 512KB L2			
	L2 Access Time	**Total L2 Area**	**L2 Power**
Ideal-CMD	101.87%	75.37%	310.28%
Practical CMD	1.58%	2.98%	1.01%

per access in larger L2 caches. Our proposed CMD mechanism with $K = 4$ generations uses 4 Bloom filters, each with $4 * N$ bits where N is the number of cache lines. It also uses 4 generation bits per line in the cache's meta-data array. The Bloom filters are used only on cache misses, so they do not affect cache latency and have minimal impact (which we account for) on power. The extra bits in the cache's meta-data array affect cache latency slightly, but this 1.5% increase in latency is unlikely to result in requiring an extra cycle for a cache hit. The overall energy impact of our CMD mechanism is minimal—it increases per-access energy by about 1%. Finally, our CMD mechanism uses less than 4% extra area in total, relative to the original area of the cache.

5.3 Classification of Coherence Misses

As we transition to multi-core processors, there is another category of misses called coherence misses. These misses occur as a result of coherence actions between private caches of various cores.

Figure 37 shows the breakdown of cache misses into coherence and non-coherence misses as we increase the number of cores from 8 to 64 for benchmarks with the most data sharing. With more cores, coherence misses represent an increasing percentage of all cache misses. As a result, performance issues related to true and false sharing become worse when more cores are used, and often become a scaling limiter. It is important to provide developers with tools and mechanisms that detect the presence of these misses,

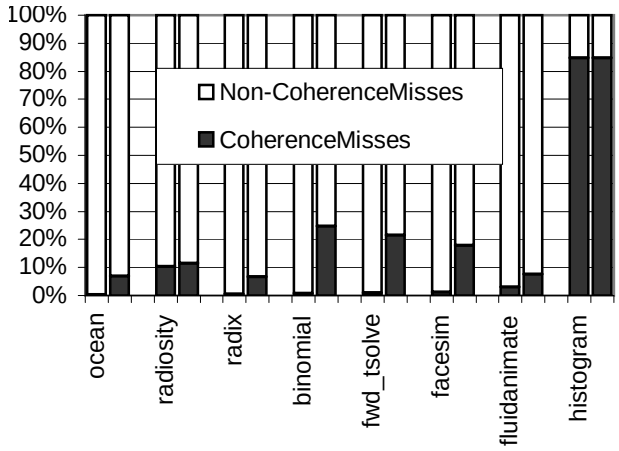


Figure 37: Breakdown of all cache misses for 8 and 64 cores.

distinguish between the two types, and identify their sources.

In this section, we first describe the relatively simple mechanism for distinguishing coherence misses from non-coherence misses, then discuss an Ideal False Sharing Detector (I-FSD) mechanism that further classifies coherence misses into those caused by false sharing and those caused by true sharing. We then contrast I-FSD with the algorithm proposed by Dubois et al. [23] and show the key differences between the two schemes. We show why I-FSD is impractical to implement in real hardware and explore several practical hardware designs for coherence miss classification.

5.3.1 Identification of Coherence Misses

Coherence misses can be distinguished from other (cold, coherence, or conflict) misses by checking if the cache block is already present in the cache. For non-coherence misses, the block either never was in the cache (cold miss) or was replaced by another block (capacity or conflict miss). In contrast, a coherence miss occurs when the block was *invalidated* or *downgraded* to allow another core to cache and access that block. Coherence misses are easily detected with a minor modification to the existing cache lookup procedure. A cache miss is detected as a coherence miss if a block does have a matching tag but does not have a valid state. Conversely, if no block with a matching tag is found, we have a non-coherence miss.

5.3.2 Ideal False Sharing Detector (I-FSD)

We use the *programmer-centric* definition of false sharing described in Chapter 2. Basically, a coherence miss is a false sharing miss if none of the memory accesses (starting with the memory access causing the miss up until the block is invalidated or downgraded or replaced) happen on data that were accessed by other cores. Coherence misses are classified into false sharing misses and true sharing misses by our I-FSD algorithm [75] shown in Figure 38.

Coherence miss classification involves two distinct phases namely:

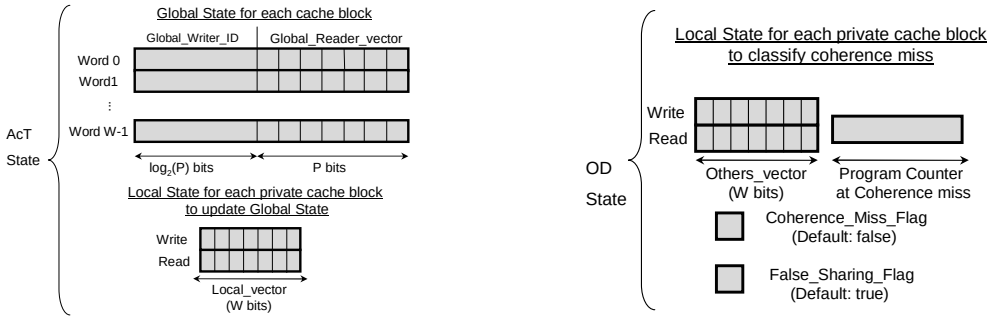
- *Access Tracking (AcT)*: From the time, a cache block is invalidated or downgraded in a core's cache, until the time when coherence miss happens, we need information about which words were read from or written to by other cores.

- *Overlap Detection (OD)*: From the time of coherence miss until when the block is invalidated/downgraded/replaced, we need to determine whether an access to any word in the block overlaps with an access by another core. True sharing access happens when a memory word is either i) written and read by two different cores or ii) written by two different cores. If we do not detect any true sharing access, then the coherence miss is a false sharing miss. Note that read by two different cores do not cause invalidation or downgrade, hence coherence misses don't occur in these cases.

The two phases, namely AcT and OD, are *temporally disjoint* phases for coherence miss classification. AcT phase occurs between the time a cache block is invalidated or downgraded from a cache and when the coherence miss happens (that is, when the cache block is accessed again by the core). OD phase occurs between the time of coherence miss and when the cache block is invalidated or downgraded or replaced.

In order to track these phases accurately, I-FSD maintains the following data structures namely:

```
Input: P=total number of cores, W=number of words in a cache block
       A=current data address, B=cache block holding address A
       c=current core
```




```
ALGORITHM ACCESS_TRACKING
  IF(Coherence Miss to B)
    WAIT ON current sharers to update Global_Writer_ID and Global_Reader_vector
  IF(Write access to A)
    Local_vector_write[A]=1;
    Local_vector_read[A]=0;
  IF(Read access to A)
    Local_vector_read[A]=1;
  IF(Invalidation/Downgrade/Replacement request for B)
    FOR(each word address K in B) DO
      IF(Local_vector_write[K]==1)
        Global_Writer_ID=c;
        Global_Reader_vector={0};
      IF(Local_vector_read[K]==1)
        Global_Reader_vector[c]=1;
    DONE

ALGORITHM OVERLAP_DETECTION
  IF(Coherence Miss to B)
    Coherence_Miss_flag=true;
    False_Sharing_flag=true; //Reset on seeing first true sharing on the block
    Miss_PC=PC;
    Others_vector_write=Others_vector_read={0};
    FOR(each word address K in B) DO
      IF(Global_Writer_ID!=c AND Global_Reader_vector[c]!=1)
        Others_vector_write[K]=1;
      IF(a bit other than c is set in Global_Reader_vector)
        Others_vector_read[K]=1;
    DONE
  IF(Write access to A)
    IF(Others_vector_write[A]==1 OR Others_vector_read[A]==1)
      False_Sharing_Flag=false;
  IF(Read access to A)
    IF(Others_vector_write[A]==1)
      False_Sharing_Flag=false;
  IF(Invalidation/Downgrade/Replacement request for B)
    IF(Coherence_Miss_flag==true)
      //Output Miss_PC and False_Sharing_flag to Profiler
```

Figure 38: I-FSD Algorithm

- `AcT state` which has the following information: i) per-word[1] Global State that tracks last writer core and subsequent readers until the next write. ii) Two bit vectors for every private cache block that track read and write accesses by the local core during *OD* to update the Global State at the time of cache block invalidation/downgrade/replacement. This is needed for other cores to perform their *AcT* correctly.

- `OD state` which has the following information: i) Two bit vectors for every private cache block which records reads and writes performed by other cores to the cache block prior to the time of coherence miss. ii) Program Counter at the time of coherence miss for attribution to the program code that caused the miss. iii) Two flags, one to track whether the cache block suffered coherence miss and the other to track whether the all the accesses to the cache block are false sharing accesses.

`PROC ACCESS_TRACKING` implements the AcT phase. Information regarding every read and write access by a core is accumulated in Local_Vectors and at the time of cache block eviction, the information is transmitted to update the Global State.

`PROC OVERLAP_DETECTION` implements the OD phase. At the time of coherence miss, the information from Global State is captured into Others_vectors. On every read and write, this information is used to determine if the current memory access results in true sharing. A read access is a true sharing access if the word's last write was not done by this core and if this core has not already read the word since it was last written. A write access is a true sharing access if the word's last write was not done by this core or if it was read by another core since it was last written. A coherence miss is classified as a true sharing miss if any access, starting with the one that causes the miss and ending with the subsequent replacement or invalidation of the block, is a true-sharing access. Conversely, the miss is categorized as a false sharing miss if no true sharing access occurs in this interval.

[1]Granularity is a matter of cost-performance trade-off. We didn't observe significant coherence activity on sub-word accesses. Hence we chose word-granularity for our work. We note that byte-granularity can be achieved with the same algorithm with relatively straightforward modifications.

Table 7: Program Counters and the corresponding number of false sharing misses reported in Facesim Benchmark.

Program Counter	False Sharing Count
4cc084	9921186
4cc074	7803903
53eb34	251457
53f2ac	214323
53f5d8	201885
...	
Total=	18854631

We present an example case study for Facesim benchmark from Parsec-1.0 suite [3] to show how our I-FSD algorithm correctly captures false sharing misses. Since the existing version has already been tested and fine-tuned for performance before release, we introduce false sharing explicitly. This is done to backtrack the original efforts by programmers to identify false sharing except that they did not have access to our tool during development. We present our results to examine the effectiveness of our tool in identifying the regions of code where false sharing misses occur.

Inside the *One_Newton_Step_Toward_Steady_State_CG_Helper_II()* function, we identify a tight loop where there are two accumulator variables *rho_new* and *supernorm* whose values are computed and updated in every iteration using the input elements. Since, each thread privately owns the accumulators and are stored in a shared array, all coherence misses that result from accessing them are false sharing misses. This limits scalability to 4x on a 8-core machine.

When we run Facesim through our I-FSD tool, we obtain the results shown in Table 7. Clearly, the static instructions at addresses `4cc084` and `4cc074` are responsible for 94% of false sharing coherence misses in the program and approximately 24.5% of all cache misses. Using the *addr2line* utility from GCC toolchain [25], we identify the part of the program where these addresses are located and find that they point to lines inside the loop where the accumulators *rho_new* and *supernorm* are being updated (Figure 39). We change

```
void One_Newton_Step_Toward_Steady_State_CG_Helper_II(...)
{
 ...
 for(int i=1;i<=dX.m;i++)
 {
  dX(i)+=alpha*S(i);
  R(i)+=alpha*negative_Q(i);
  double s2=R(i).Magnitude_Squared();
  rho_new+=s2;
  supnorm=max(supnorm,s2);
 }

 //False Sharing Fix
 // Locally accumulate the counters and update later
 //for(int i=1;i<=dX.m;i++)
 //{
 // dX(i)+=alpha*S(i);
 // R(i)+=alpha*negative_Q(i);
 // double s2=R(i).Magnitude_Squared();
 // local_rho_new+=s2;
 // local_supnorm=max(local_supnorm,s2);
 //}
 //rho_new=local_rho_new;
 //supnorm=local_supnorm;

 ...
}
```

Figure 39: False Sharing Misses in Facesim benchmark.

the loop to accumulate the results locally into *local_new_rho* and *local_supernorm* and update shared variables *new_rho* and *supernorm* after the loop is complete. This avoids the false sharing coherence misses on these accumulators and results in near-linear (near-8x) speedup on an 8-core machine.

5.3.3 Comparison with Dubois' scheme

To our knowledge, Dubois et al. [23] was the first proposal to classify coherence misses as essential (true sharing) and non-essential (false sharing). They propose separate algorithms for different types of cache coherence protocols. We use the invalidate-based protocol (MESI) implementation described in [23] for our comparison. This implementation maintains a *stale bit* for every cache word. When a write happens on a word, the caches that currently have the word clear the stale bit to indicate that a new value is produced. A subsequent read of this word denotes consumption of "fresh value" produced by a write and is treated as true sharing. Also, the stale bit is set to denote that any further read (without an

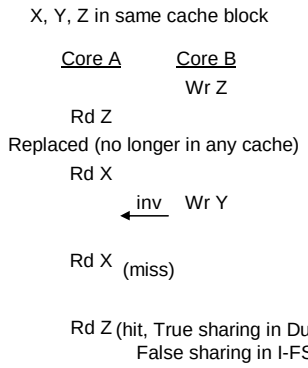


Figure 40: Accurate classification requires us to keep access information even for blocks that are no longer in any cache.

intervening write) will only consume a stale value.

There are two key differences between I-FSD and Dubois scheme. First, writers (producers) never classify coherence misses and all write misses are assumed to be non-essential misses. Hence, a portion of true sharing misses that can classified on the producer's side are classified as non-essential misses in Dubois' scheme. Whereas, I-FSD classifies coherence misses both on the producer and consumer sides. In a *single producer-multiple consumer* pattern (See Gauss-Siedel example in Section 2.4.2) and *multiple producer-multiple consumer* pattern (See histogram example in Section 2.4.2), coherence misses are highly likely to happen in multiple cores (producer and the respective consumers). Dubois' scheme could underestimate the effect of false sharing in such situations and the programmer might consequently ignore or oversee the lines of code involved in such patterns because of lack of understanding of its impact.

Second, on every cache block replacement, the stale bits are cleared. This is because, the *staleness* of values tracked by the consumers are no longer relevant by virtue of being replaced by another cache block. This results in loss of history about the "staleness of values" (information that a consumer had already read the value from the cache word). When the cache word is subsequently read by the core, the stale bit is zero and hence, it is treated as true sharing by Dubois' scheme. Figure 40 shows an example where the cache block having items X, Y and Z has been replaced by both cores A and B and when core A

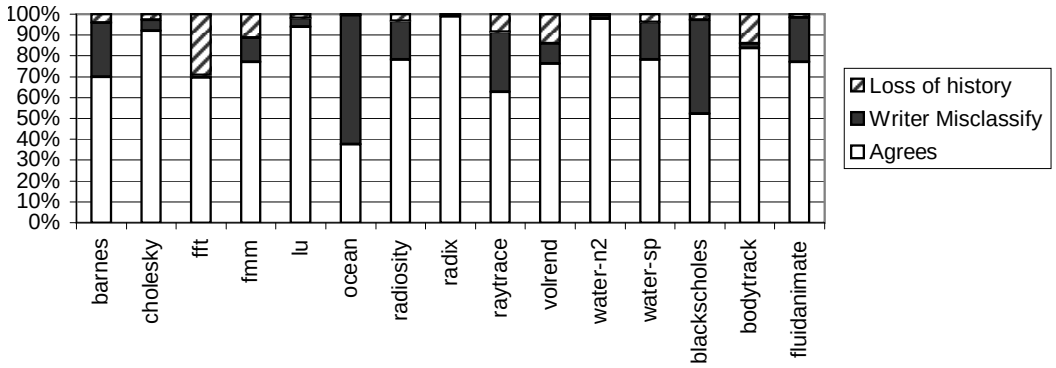


Figure 41: Comparison between I-FSD and Dubois' scheme. Each bar has three portions-bottom portion shows the percentage of times there is agreement between two schemes, middle portion shows percentage times Dubois' scheme misclassifies as false sharing (non-essential) on the producer side and the top portion shows the percentage times Dubois' scheme misclassifies as true sharing because of loss of history regarding *staleness* of values.

suffers a coherence miss, Dubois' scheme classifies "Read Z" as true sharing access. This is because, the information regarding the "staleness" of value Z has been lost during cache block replacement. However I-FSD remembers that the value of Z had already been read by core A and would correctly classify the access as false sharing.

Figure 41 shows the results of our experiments. Each benchmark has three portions. The bottom portion of each bar shows the percentage of coherence miss classifications that both Dubois' and I-FSD schemes agree. In many benchmarks, the two schemes agree atleast 75% of the time except fft, ocean, raytrace and blackscholes. The middle portion of each bar shows the percentage of times the writer classifies coherence misses as non-essential (false sharing) in Dubois' scheme whereas I-FSD detects true sharing on the writer side. A large portion of such misclassifications occur in benchmarks such as ocean (62%), blackscholes (45%), raytrace (28%) and fluidanimate (21%). The top portion of each bar shows the percentage of times Dubois' scheme detects essential (true sharing) whereas I-FSD detects false sharing. Such cases occur when *stale bits* are cleared on cache block replacement. This results in loss of history information regarding "staleness" of values that are already being read. Such misclassifications are frequent in a few benchmarks such as fft(29%), volrend(14%) and bodytrack (14%). I-FSD correctly detects false sharing

because it maintains the access history information in global state (similar to a central repository).

5.3.4 Suitability of I-FSD for On-Line Miss Classification

There are a number of problems that make the Ideal False Sharing Detector unsuitable for on-line implementation needed to drive hardware performance counters or other hardware performance debugging and attribution mechanisms. The most significant of these problems are:

1. The I-FSD has a high implementation cost which does not scale well. The *per-word* Global State for the I-FSD structure shown in Figure 38 is $P + log_2 P$ where P is the number of cores. With 4 cores and 32-bit words this state represents a 19% storage overhead, and with 32 cores the storage overhead is already 116%.

2. The state needed for the I-FSD can be very large (requires keeping state for all words ever touched). It may be tempting to not keep track state for a word that is no longer present in any core's cache. However, information about currently evicted blocks might be relevant for classifying future coherence misses to these blocks (See example in Figure 40).

3. Changes are needed to the underlying cache coherence protocol and extra network traffic is required to update the I-FSD state and/or propagate it to the cores that need it to classify their coherence misses. In particular, Global State information for a memory word should be kept in a central repository (could also be stored in the last level shared cache or memory). As seen in Figure 38, the coherence protocol needs to be modified to trigger reads and writes of the global state information at appropriate times. To enforce that, the central repository needs to ensure updates sent by the cores replacing/invalidating/downgrading the cache block are reflected in the Global State before the core suffering the coherence miss reads the information from the central repository and updates its local state to perform *OD*.

4. A coherence miss may be classified as a true or false sharing miss many cycles after the instruction causing the miss has retired from the core's pipeline. This delay in classification makes it more difficult and costly to attribute false and true sharing misses to particular instructions, especially if performance counters support precise exceptions for events, such as PEBS (Precise Event Based Sampling) mechanism available in recent Intel processors [2]. For accurate attribution, the PC of the instruction that caused a coherence miss must be kept until the miss is eventually classified. This increases the cost of the classification mechanism, and also makes it more difficult for profilers to extract other information about the miss (e.g., what was the data address or value) [2].

To provide a realistically implementable scheme for on-line classification of coherence misses, we need to overcome some (preferably all) of the above problems, while sacrificing as little classification accuracy as possible. When choosing which aspects of classification accuracy to sacrifice, we keep in mind the primary purpose of our classification mechanism: giving the programmer an idea of how much performance is affected by true and false sharing cache misses, and pinpointing the instructions (and from there, lines of code) that suffer most of these misses. Armed with this, the programmer should be able to make an informed decision about how best to reduce the performance impact of these misses.

5.3.5 Practical False Sharing Detectors

We propose and explore a range of design choices that trade-off accuracy for lower hardware costs. Section 5.3.4 enumerated a number of issues that limit I-FSD from being implemented in real machines. The key to overcoming these problems is to reduce the amounts of both global and local state. To achieve this, we make use of the fact that there are two distinct phases in Coherence miss classification namely *AcT* and *OD* (Section 5.3.2). The data structures needed by these two phases are independent and hence the mechanisms needed for the two phases can be designed independently. We call these

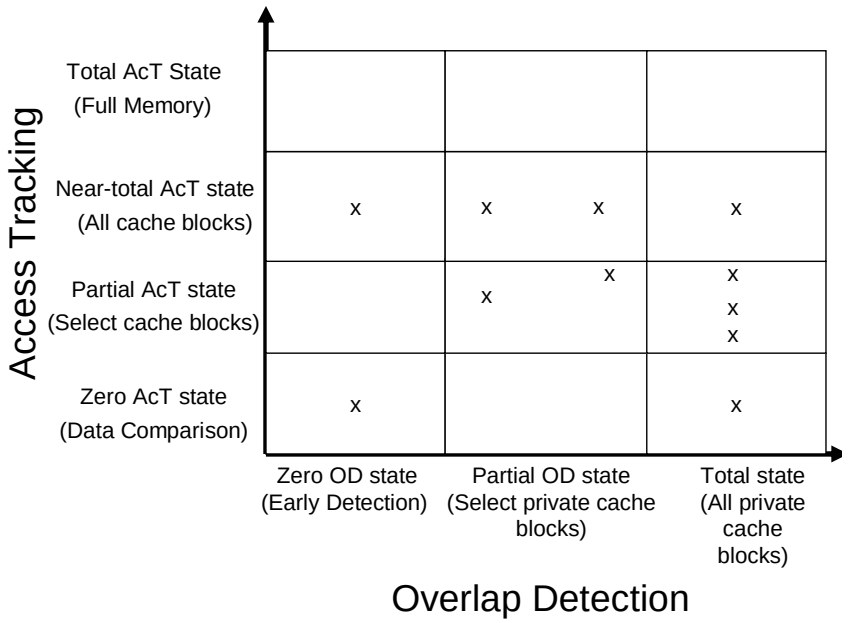


Figure 42: Design Space for coherence miss classification mechanisms. Gradations of AcT state is on the vertical axis and OD state is on the horizontal axis. Hardware costs increase as we move up or towards the right. Design points marked 'x' are examined in our evaluations.

mechanisms as practical FSD.

As shown in Figure 38, the Global State along with local access vectors (that track reads and writes by the local core to eventually update the global state) are needed to implement the *Access Tracking* phase correctly. We call this `AcT state`. Rest of the local state (namely Read and Write vectors of others, flags and the program counter at coherence miss) are used to implement the *Overlap Detection* phase. We call this `OD state`. Figure 42 shows the design space of coherence miss classification mechanisms defined by how much state is kept for each of these two phases.

5.3.6 Design Choices for Access Tracking

The most expensive AcT scheme (top row) maintains the AcT state for the entire memory throughout the program execution. We call this *Total AcT State*. As described in Section 5.3.4, the amount of global state grows proportional to the number of cores and the memory overhead increases at a super-linear rate with the number of cores. Hence this solution is

impractical in real hardware.

The next most aggressive AcT implementation (second from top) maintains AcT state only for cache blocks in the on-die caches. We call this *Near-Total AcT state*. This scheme can only lead to misclassification of coherence misses to blocks that are evicted from all of the on-die caches between an invalidation/downgrade and the subsequent coherence miss. Evictions of such lines are highly likely to be a small fraction of the evictions, and thus this event should be quite uncommon. The state can be kept with the lowest level shared cache, for systems that have one. For chips with only private caches, and a separate mechanism for maintaining coherence such as a directory, the AcT state can be kept along with the coherence state for the core.

The third most aggressive AcT implementation (third from top) maintains Global State only for a limited number of cache blocks in the on-die caches. We call this *Partial AcT state*. This scheme can lead to misclassification of coherence misses if the cache block currently suffering coherence miss is not tracked globally. In order to select the subset of cache blocks for tracking global state, as a first order approximation, we could choose only blocks that currently occupy the last level private caches that are kept coherent. Furthermore, not all blocks in coherent caches are involved in sharing. Hence, we have opportunities to further restrict the number of entries to be tracked. To implement Partial State, we maintain a global pool where entries are allocated on demand to track select cache blocks. An entry is created when a cache block in any of the coherent caches is either invalidated or downgraded as they are likely to suffer from coherence misses. Subsequent accesses by various cores to the block are tracked and updated as described in I-FSD algorithm. Since the global pool holds a limited number of entries, it needs a policy to replace existing entries to make room for incoming blocks. We pick a modified version of pseudo-LRU replacement policy called NkMRU(Not 'k' Most Recently Used) replacement policy. Basically, it chooses a random entry for replacement that is not among the 'k' most-recently-used cache blocks. It should be noted that not all invalidated or downgraded cache blocks will eventually suffer

coherence misses. Hence promoting every incoming block into the k most-recently-used set could result in losing valuable information about blocks that frequently suffer from coherence misses. To prevent this effect, we promote a cache block entry into the k most-recently-used set only when the cache block actually suffers a coherence miss in one of the caches.

The final and least aggressive AcT implementation (bottom row) does not maintain any AcT state and *infers* false and true sharing locally by comparing the stale value of the cache block with the incoming values. This technique of detecting false sharing through data comparison was used by Coherence Decoupling [33] to save cache access latency due to false sharing misses. If the data value has changed, then true sharing is detected, otherwise, the coherence miss is attributed to false sharing. We call this *Zero AcT state*. This scheme could misclassify in two situations: i) Silent stores [41] do not change the data value and hence, it is impossible to detect true sharing in such situations. However, classifying silent stores itself is a murky area. Whether they contribute to true or false sharing depends on specific situations. For example, lock variables have an initial value of zero. A core grabs a lock by setting it to one. Later when the lock is freed, it deposits a value of zero back. An external core does not see change in lock value and effectively sees a value of zero before and after the core operated on the lock. Even though lock sharing is a form of true sharing, detecting false sharing through data comparison would miss this effect. At the same time, other silent writes that simply do not communicate any new value can be eliminated through algorithmic changes, ii) While readers would be able to detect change in data value due to an external write, writers do not detect true sharing as readers do not modify data. As long as coherence misses are classified on the consumer side correctly, the programmer would still get an accurate picture of false sharing effects in the program. Note that, there is no AcT state and therefore, to enable OD, external writes inferred through data comparison should be recorded. This scheme basically needs a bit vector where bits get set to indicate change in data values. Other structures used in I-FSD such as flags to indicate

coherence miss and false sharing as well as program counter counter that caused the miss are still needed to correctly perform OD.

5.3.7 Design Choices for Overlap Detection

We explore various design choices for OD shown in Figure 42. The most aggressive OD implementation (right column) maintains local state for all blocks in the coherent private caches. We call this *Total OD state*. As a result, it yields highest accuracy by capturing all local state needed to accurately perform OD. However, it is expensive to maintain local state bits for every cache block.

The next most aggressive OD implementation (second column) maintains local pool to track local state of a limited number of cache blocks that suffer coherence misses. We call this *Partial OD state*. This implementation does not contribute to any misclassification of coherence misses. However, a coherence miss may not be tracked when there are not enough entries in the local pool. The local pool entry is freed from the local pool after classifying the coherence miss. Since, at any given time, not all cache blocks are involved in sharing, a local pool (similar to global pool) works well for tracking local state in private caches. If the AcT involves maintaining partial state, then an entry is created in the local pool only if the global pool tracks the corresponding cache block.

The least aggressive OD implementation (left column) does not maintain local state corresponding to OD for any cache block. We call this *Zero OD state*. At the time of coherence miss, it simply observes whether the data address causing the miss suffers true or false sharing and classifies the miss at this point. This has the potential to overestimate false sharing misses in a program because early classification would ignore true sharing access on a block that might happen much after the point of coherence miss.

5.3.8 Non-primary Private Caches

Local state is relatively easy to update and check in a primary (L1) cache where all memory accesses are visible to the cache controller. In lower-level caches, only cache line addresses

are visible. As a result, in systems where multiple levels of cache may be involved in coherence (e.g., with private L2 caches), information from a private non-primary (L2) cache should be passed to the primary cache (L1) when it suffers a cache miss. The L1 cache then keeps track of the flags that track coherence miss and false sharing for the L2 cache and forwards the value of these flags back to L2 when the block is replaced from the L1 cache. Although in this scenario we have caches directly communicating access related information, it only happens between different levels of private caches for the same core, and only on a coherence miss or eviction.

5.3.9 Other issues

The relatively simple mechanism to detect coherence misses described in Section 5.3.1 can err in two ways. First, on power-up or after a page fault, the state of a block's state is set to invalid, but the tag need not be initialized. This could sometimes result in a tag accidentally matching to that of a requested block. However, this would be quite rare and random enough that they do not attribute in significant numbers to any particular piece of code. Second, cache replacement policy could prioritize replacement of invalidated blocks, which can destroy the evidence of a coherence miss. For highly contended blocks involved in sharing patterns, there is little time for the block to be replaced between the invalidation and the subsequent miss, so replacement priority is unlikely to obscure enough coherence misses to hide a scalability problem. It should also be noted that the inaccuracy caused by replacement priority is very costly to avoid: it requires the cache to track the tags of blocks that *would be* in the cache were it not for replacement priority, e.g. using a duplicate set of tags.

For machines with shared bus interconnect and a shared last-level cache, the AcT state can easily be maintained in the shared cache. However, with the growing popularity of technologies such as HyperTransport [34], the interconnect could be high-speed, point-to-point communication links between the cores. In these cases, we maintain the global AcT

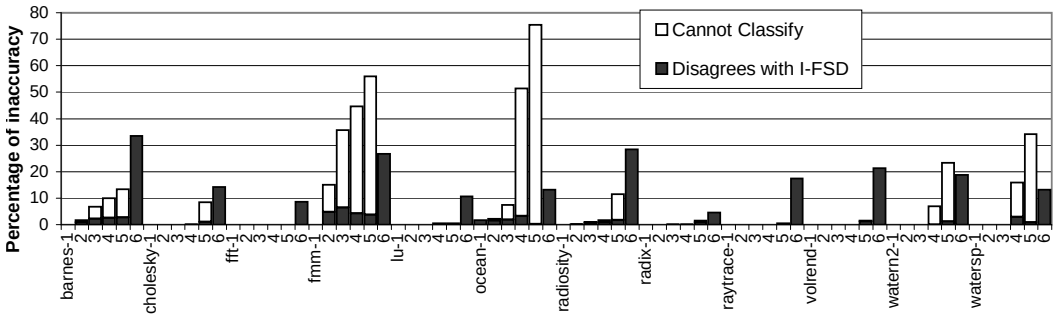
state for a block in its home node or tile, along with existing coherence state that is already part of the directory. For all dirty block (cache blocks that had write accesses from the local core) replacements, the AcT state can be easily updated by piggybacking on existing writeback messages. However, for clean block (cache blocks that had read accesses only) replacements, the reader information is sent to the home node to update the AcT state using an extra message.

5.3.10 Evaluation

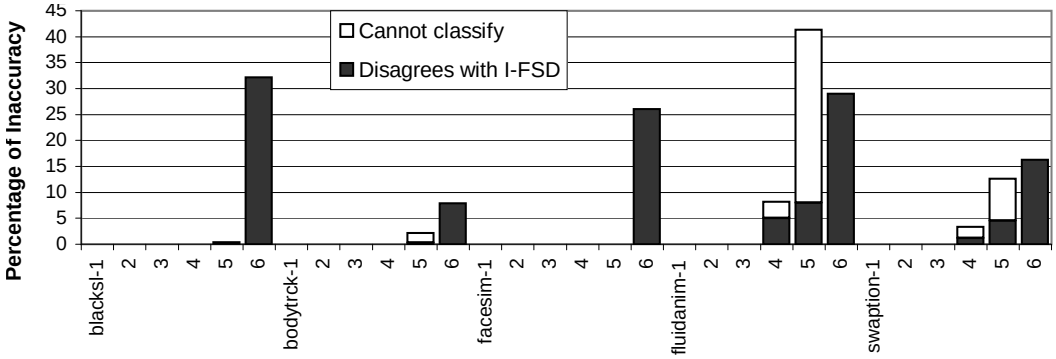
In our experiments, we use the same configuration used in Section 5.2.2 and measure coherence misses in large L2 private caches to maximize the number of coherence misses and hence study our approximate schemes. To examine how the approximations in practical schemes affect their accuracy, we measure the percentage of coherence misses that are not correctly classified by the particular scheme. We identify two aspects of inaccuracy namely– a) the practical scheme can disagree with the I-FSD scheme, that is, practical scheme reports false sharing while I-FSD detects true sharing and vice versa and b) the practical scheme may not be able to classify true or false sharing due to lack of information from the AcT state or OD state. This results from maintaining partial state for AcT and OD.

Figures 43(a) and 43(b) show the accuracy results for a range of practical schemes that maintain total OD State and varying amounts of AcT state. Each benchmark is evaluated for six different schemes (from left to right) namely Near-Total AcT state (where AcT state is maintained for every cache block with a total of 256k entries), Partial AcT state with four different configurations where the global pool maintains has 16384(16k), 4096(4k), 1024 (1k), 64 entries respectively and finally Zero AcT state, where false and true sharing are inferred locally through data comparison.

Near-Total AcT state achieves perfect accuracy in all of our benchmarks expect ocean where a small percentage ($<1\%$) of coherence miss classifications disagree with I-FSD.



(a) Splash-2 Benchmarks



(b) Parsec Benchmarks

Near-Total	Partial (16k)	Partial (4k)	Partial (1k)	Partial (64)	Zero AcT
4.36%	1.8%	1.79%	1.65%	1.57%	0.00%

(c) Area Overhead of AcT state as a percentage of on-die cache area. Total OD state adds 7.35% area overhead to on-die caches.

Figure 43: Accuracy versus cost trade-offs for practical schemes having total OD state and varying amounts of AcT state. For each benchmark, the six bars (from left to right labeled 1 through 6) represent Near-Total AcT state, Partial AcT state with 16k, 4k, 1k, 64 entries and Zero AcT state.

This inaccuracy results out of loss of AcT state that happens over very long time intervals. Partial AcT states with 16k and 4k entries capture enough accuracy with $<$10% of coherence misses being misclassified in almost all benchmarks except fmm and water-sp. Partial AcT states with 1k and 64 entries do not have not enough entries to accommodate all the blocks involved in sharing. From the attribution information, we find that a relatively few static instructions are involved in a large number of coherence misses. For instructions that infrequently suffer coherence misses, information about cache blocks they use are replaced from the global pool and hence introduce inaccuracy in these benchmarks. Our experiments show that a small percentage ($<$5%) inaccuracy results from loss of state on certain reads

and writes performed by certain cores. Even when the Partial AcT state has information for the cache block, an intervening replacement in the global pool might result in loss of access history about reads and writes performed by cores on that block. On the other hand, there are situations where inaccuracy results from unavailability of Partial AcT state. For such situations, there are two possible solutions– 1) For static instructions of interest (those that suffer coherence misses frequently), state about cache blocks they use can be tracked by better sampling techniques 2) Use Zero AcT (data comparison) technique as a fall-back mechanism to detect false sharing. Finally, Zero AcT with Total OD state incurs inaccuracy in the range of 4.5%(radix) to 33.5%(barnes). We find that the writer cores being unable to detect reads by other cores is a significant source of misclassification. The other major causes are silent writes in dynamic scheduling code used by many PARSEC benchmarks as well Radiosity in Splash-2 benchmark suites. From attribution experiments, we find that top ten offending static instructions still match in many benchmarks, although they have differences in reporting the numbers of false and true sharing misses to those offenders.

Figure 43(c) shows the area cost of our practical schemes with different AcT state as a percentage of total on-die cache area. The most aggressive Near-Total AcT scheme keeps the global AcT state for all words and maintains local OD state for the entire cache. As a result, it incurs upto 11.7% (4.36% for global state) area overhead compared to total area of the on-die caches used in our experiments. For all of the Partial AcT state schemes, the local state kept in each private cache dominates the cost. Thus reducing the amount of AcT state has little impact on area overhead. The least expensive Zero AcT scheme incurs 7.35% area overhead on on-die caches just to maintain the local OD state.

Overall, from the above experiments, we find that accuracy of partial AcT schemes is sensitive to the amount of AcT state, but area overhead is not very sensitive to the amount of AcT state, except for Zero AcT. As a result, a good design of a partial AcT scheme is one that maximizes accuracy by providing enough entries (eg. 4k or more entries).

We perform additional experiments to provide insight into the various AcT schemes.

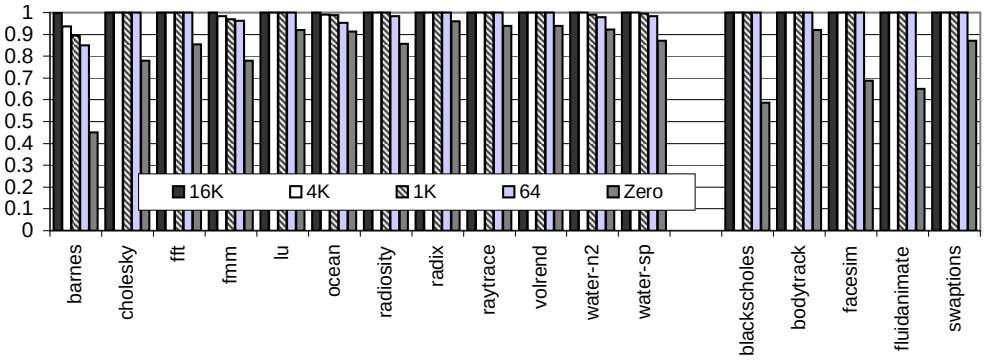


Figure 44: Correlation coefficient between number of false sharing misses suffered by static instructions in Partial AcT, Zero AcT schemes against Near-total AcT. Samples of static instructions are chosen by Partial AcT schemes.

The design points that perform Partial AcT classify coherence misses for cache blocks whose information is available in global state. Effectively, they sample a subset of coherence misses that are actually classified by Near-Total AcT. We study the correlation coefficient between the number of false sharing misses for the static instructions that are chosen as samples by the Partial AcT schemes. A correlation of +1 means that false sharing misses reported by Partial AcT are similar or proportional (equivalent modulo scaling) to the false sharing misses reported by Near-Total AcT for the chosen samples. A correlation of 0 means that they are uncorrelated (or independent of each other). Figure 44 shows the results of our experiments. We see that all benchmarks except barnes, fmm and ocean show high correlation ($>$0.99) even for 64 entries. This shows that even though the Partial AcT schemes perform classification only on a subset of static instructions causing coherence misses, we still get a very good accuracy from Partial AcT states. Zero AcT exhibits poor correlation against Near-total AcT with correlation coefficient as low as 0.45 in barnes benchmark.

We conduct experiments to study the differences between Near-Total AcT and Zero AcT and hence, verify our implementation. As first step, we modify Near-Total AcT to ignore reader information on the writer side. This is similar to Zero AcT not being able to

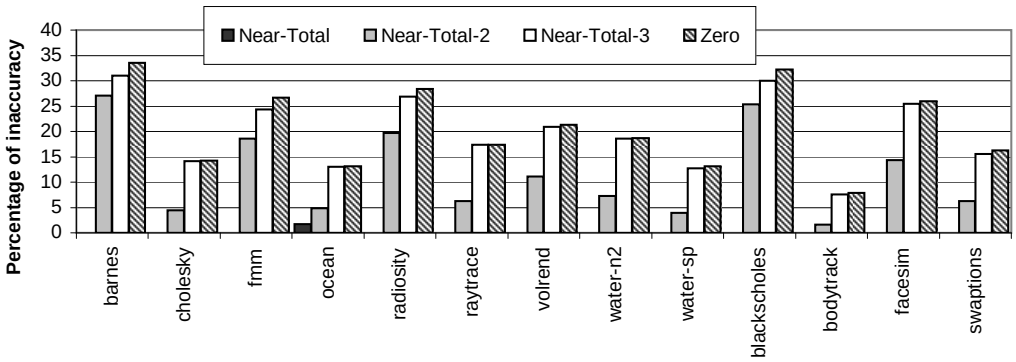
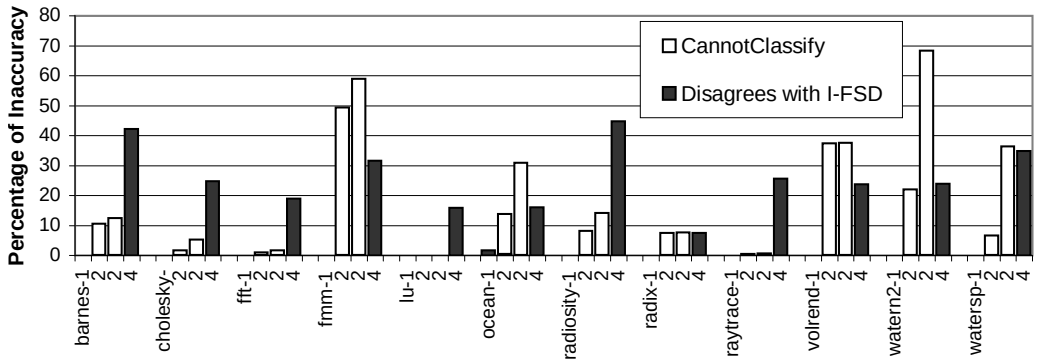


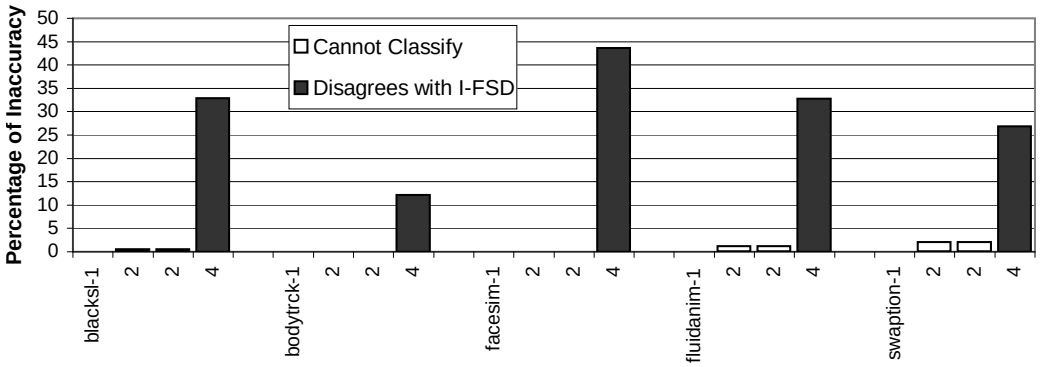
Figure 45: Percentage of inaccuracy for Zero AcT and different implementations of Near-Total AcT. For each benchmark, there are four bars- Near-Total, Near-Total-2 (does not consider reader information on the writer side), Near-Total-3 (does not consider silent writes and ignores reader information on the writer side) and Zero AcT.

detect reads on the writer side to declare true sharing. We call this implementation as Near-Total-2. We then modify Near-Total-2 to ignore silent writes. This is also similar to Zero AcT not being able to detect writes that do not change values of memory locations. We call this implementation as Near-Total-3. Figure 45 shows the result of our experiments where we measure the percentage of inaccuracy for Near-Total AcT, modified implementations of Near-Total AcT and Zero AcT. In all benchmarks (except blackscholes), there is a noticeable increase in percentage of inaccuracy once we ignore reader information on the writer side for coherence miss classification (Near-Total-2). As we further ignore silent writes from consideration (Near-Total-3), the percentage of inaccuracy almost equals to Zero AcT in all benchmarks except barnes, fmm and blackscholes. On further investigation, we find that these benchmarks have a small percentage of idempotent writes (writes that change values and deposit the old values back between coherence misses). In these cases, Near-Total AcT would classify as true sharing while Zero AcT would detect false sharing because it does not detect change of data values.

Figures 46(a) and 46(b) show the accuracy results for a range of practical FSD schemes that maintain Near-total AcT state and varying amounts of OD state. We evaluate for four different schemes for each benchmark (from left to right) – Total OD state (where OD state is maintained for every cache block with a total of 4k entries for each private cache), Partial



(a) Splash-2 Benchmarks



(b) Parsec Benchmarks

Total OD	Partial(1k)	Partial(256)	Zero OD
7.35%	3.81%	3.66%	0.0%

(c) Area Overhead of OD state as a percentage of on-die cache area. Total global AcT state adds 4.36% area overhead to on-die caches.

Figure 46: Accuracy versus cost trade-offs for practical schemes having Near-Total AcT state and varying amounts of OD state. For each benchmark, the four bars (left to right labeled 1 through 4) represent Total OD state, Partial OD state with 1k, 256 entries and Zero OD state.

OD state with two different configurations where the local pool maintains has 1024(1k) or 256 entries and finally Zero OD state (where false and true sharing are declared at the time of coherence miss).

Total OD state achieves the highest accuracy. However, Partial OD states exhibit slightly different behavior than Partial AcT states. From the figures, it should be noted that Partial OD states predominantly suffer from one type of inaccuracy with respect to I-FSD: inability to classify coherence misses when the local pool has too few entries. Our attribution experiments show that cache blocks that frequently suffer coherence misses are

still accurately captured by Partial OD state configurations, even those with reduced number of entries. Finally, Zero OD with Near-Total AcT state incurs inaccuracy in the range of 7.5%(ocean) to 45%(radiosity). Early classification of coherence misses results in missing true sharing accesses that occur after the time of coherence miss (Section 2.4.1). Hence this scheme yields little value (accuracy) for the amount of global state maintained for each word. In contrast, at the opposite end of the design space, (Total OD state, Zero AcT state) yields higher accuracy at lower cost than this scheme (Section 5.3.6). This indicates that Zero OD schemes only make sense for extremely low cost designs i.e, only for combinations with Zero AcT.

Figure 46(c) shows the area cost incurred by practical schemes with different OD states as a percentage of total on-die cache area. Maintaining partial OD state for 1k entries incurs about 3.8% area overhead to on-die caches and for 256 entries, it costs about 3.66% additional cache area. For the Zero OD state, the area overhead of 4.36% is needed to keep global AcT state.

Overall, from the above experiments, accuracy drops marginally when moving from Total OD state to Partial OD states while we lose accuracy sharply when moving from Partial OD states to Zero OD states. Because the associated area overheads are similar for Zero OD and Partial OD, configurations with partial OD are preferable over Total OD ones (nearly similar accuracy for less cost).

Partial OD states sample a subset of coherence misses and classify them because of limited history information maintained by them. Effectively, these design points perform sampling and determine true and false sharing. We conduct experiments to study the correlation between the number of false sharing misses reported in both the Total OD and the corresponding Partial OD state for the static instructions that are chosen as samples by the Partial OD schemes. Figure 47 shows the results of our experiments. We see that all benchmarks except fmm show high correlation ($>$0.99) even for 256 entries. This shows that even though the Partial OD schemes perform classification only on a subset of static

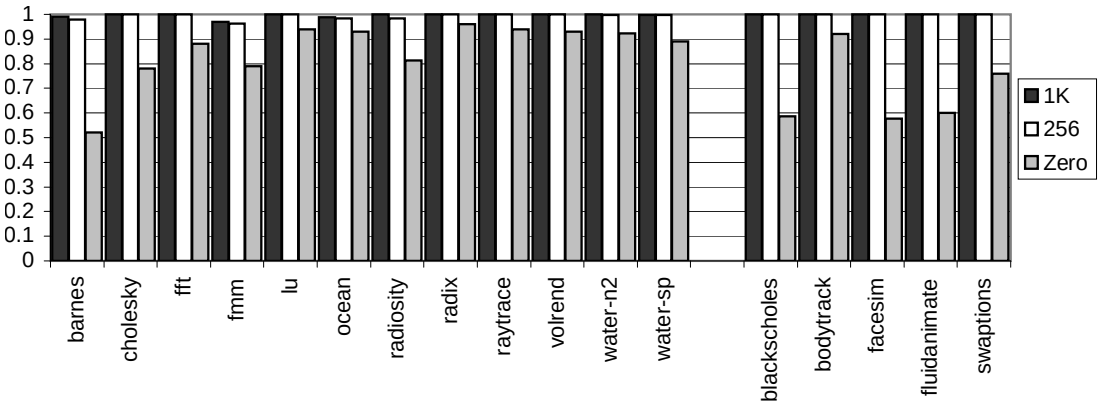


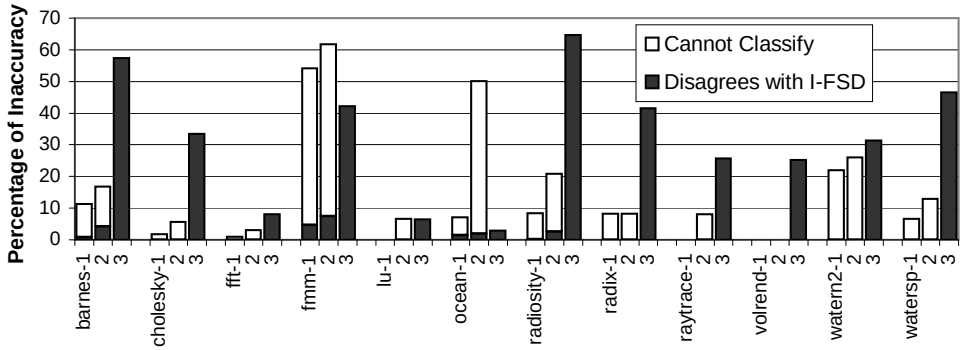
Figure 47: Correlation coefficient between number of false sharing misses suffered by static instructions in Partial OD, Zero OD schemes against Total OD. Samples of static instructions are chosen by Partial OD schemes.

instructions causing coherence misses, we still get a very good accuracy for those samples from Partial OD states. Zero OD shows poor correlation against Total OD with correlation coefficient as low as 0.52 in barnes benchmark.

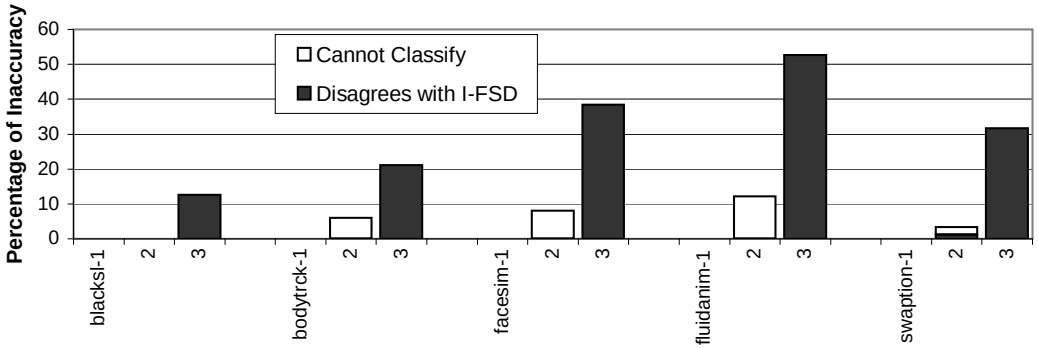
Even though, the preceding discussion appears to categorize the two phases of coherence miss classification namely *Access Tracking* and *Overlap Detection* into discrete design points, in reality, we get a continuous spectrum of design points along both the two axes. Partial State along both directions provides a continuum between maintaining zero state and full state. There are a wide range of possibilities in this design space. The configuration that is most appropriate for a user is to identify the region in the partial state that helps capture the entire picture for the programmer that would otherwise be obtained through maintaining total global and local states.

Figures 48(a) and 48(b) show the accuracy results for a range of FSD schemes that are relatively low cost and can be supported in real hardware. For each benchmark, we show three different schemes (from left to right) – i) (Partial(1k) OD, Partial (16k) AcT), ii) (Partial(256) OD, Partial(4k) AcT) and iii) (Zero OD, Zero AcT).

From prior experiments, we have seen that partial states offer reasonable accuracy while achieving lower costs. We pick partial AcT states that preserve relatively high accuracy (global pool having 16k and 4k entries) and evaluate them in combination with the partial



(a) Splash-2 Benchmarks



(b) Parsec Benchmarks

Partial(1k) OD Partial(16k) AcT	Partial(256) OD Partial(4k) AcT	Zero OD Zero AcT
5.61%	5.45%	0.0%

(c) Area Overhead of OD and AcT states as a percentage of on-die cache area.

Figure 48: Accuracy versus cost tradeoffs for certain low-cost practical FSD schemes. For each benchmark, the three bars (left to right labeled 1 through 3) represent (Partial (1k) OD, Partial (16k) AcT), (Partial(256) OD, Partial(4k) AcT) and (Zero OD, Zero AcT) points in the design space.

OD state that has 1k and 256 entries in the local pool respectively. Our experiments show that this combination has similar accuracy to the Partial AcT states with 16k and 4k entries respectively in combination with Total OD. Benchmarks such as fmm, ocean and water-n2, which suffer additional inaccuracy from reduced number of OD entries. Finally the (Zero OD, Zero AcT) design point is the least accurate but is almost free in terms of cost. We find that this scheme has a highest misclassification percentage in many benchmarks with a maximum upto 65%(radiosity). This design point was used in Coherence Decoupling [33] to perform value speculation and hide cache miss latency due to false sharing. Although this design point is sufficient for speculation (where there are recovery mechanisms), it might produce misleading results for the programmer during performance debugging.

Figure 48(c) shows the area cost incurred by FSD schemes as a percentage of total on-die cache area. Partial schemes incur relatively modest area overheads of about 5.5% , a major portion (about 70%) of which is incurred by maintaining local pools for every private cache in our 64-core configuration. The (Zero OD, Zero AcT) scheme can be implemented with minimum hardware modification such as changing the cache controller to perform data comparison on a coherence miss. Hence, this scheme has near-zero cost. Overall, we find that partial schemes can offer relatively good accuracy compared to I-FSD with reasonably modest costs.

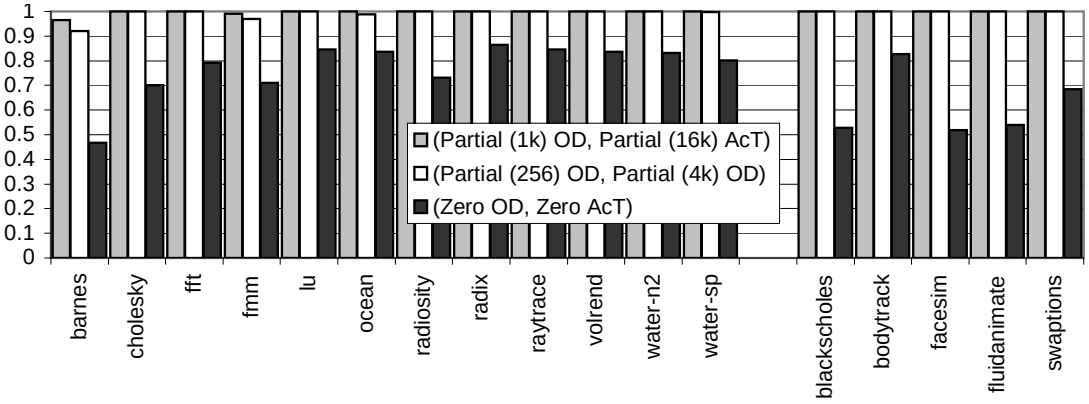


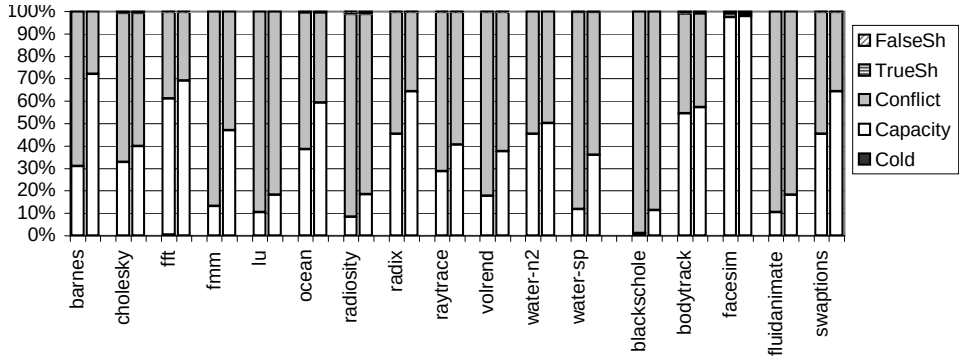
Figure 49: Correlation coefficient between number of false sharing misses suffered by static instructions in various schemes against (Total OD, Near-Total AcT). Samples of static instructions are chosen by Partial state schemes.

Schemes with partial OD and AcT state sample a subset of coherence misses and classify them because of limited history maintained both globally and locally. Effectively, these design points perform sampling and determine true and false sharing. We conduct experiments to study the correlation between the number of false sharing misses reported in both the (Total OD, Near-Total AcT) and the corresponding Partial state scheme for the static instructions that are chosen as samples by the Partial state schemes. Figure 49 shows the results of our experiments. We see that all benchmarks except barnes and fmm show high correlation (>0.99) even for 256 entries. This shows that even though the Partial state schemes perform classification only on a subset of static instructions causing coherence misses, we still get a very good accuracy for those samples from Partial states. (Zero OD, Zero AcT) shows poor correlation against (Total OD, Near-Total AcT) with correlation coefficient as low as 0.468 in barnes and 0.51 in facesim benchmarks.

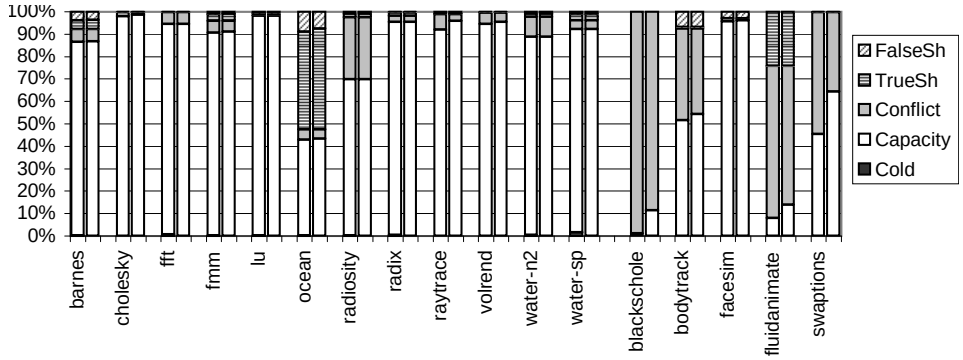
5.4 Comprehensive classification

In this section, we perform comprehensive classification of cache misses using ideal and practical schemes described in Section 5.2 and 5.3. We use I-CMD and I-FSD schemes to show how ideal off-line schemes would classify cache misses and compare them against practical schemes with reasonable cost-accuracy trade-offs. We use practical CMD with 4 generations for classifying replacement misses and (Zero AcT, Total OD) for classifying coherence misses.

We perform experiments to study the cache miss classification both in 32 KB, 4-way private L1 caches (results shown in Figure 50(a)) and 512 KB, 16-way private L2 caches (results shown in Figure 50(b)). Each benchmark has two bars, the first bar shows the cache miss breakdown according to ideal offline schemes and the second bar shows the breakdown of cache misses in the practical schemes. In L1 caches, coherence misses form a negligible (<1%) fraction of the overall cache misses. Among the replacement misses, misclassification of conflict misses as capacity misses arises from "near-capacity" conflict



(a) private L1 cache



(b) private L2 cache

Figure 50: Cache miss classification for smaller (32 KB) L1 and larger (256 KB) L2 caches.

misses (See Section 5.2.2). In L2 caches, coherence misses form a significant portion in certain benchmarks like barnes (8%), ocean (52%) and fluidanimate (25%). Conflict misses represent less than 10% of overall cache misses except in a few benchmarks like radiosity, blackscholes, bodytrack, fluidanimate and swaptions. This phenomenon is largely because of increased associativity in L2 compared to L1 caches [29]. Finally, cold misses represent negligible (<0.1%) of cache misses in both L1 and L2 caches.

Figure 51 shows the effect of different types of cache misses on pipeline stalls. When a load or store instruction occupies the head of Re-Order Buffer (ROB) and is unable to retire because of a cache miss, we attribute the pipeline stall to that cache miss. Our experiments show that capacity misses incur the highest average with 8.5 stall cycles per miss and true sharing misses have the least average with 7.14 cycles per miss. However, we note that all types of cache misses incur similar penalties in terms of pipeline stalls. This confirms our

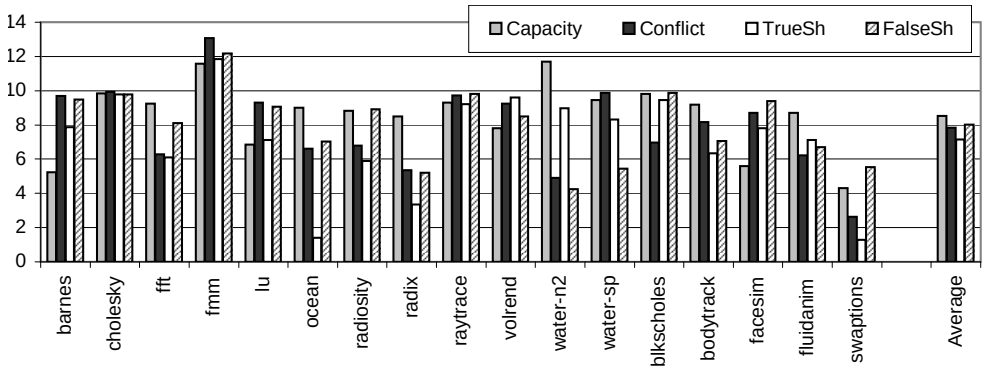


Figure 51: Average per-miss pipeline stall cycles incurred by different types of cache misses.

intuition that the symptoms of cache misses are often similar, whereas the fixes needed to address them can be very different, and often, depend on the type of the cache miss.

5.5 Related Work

The classification of uniprocessor cache misses into compulsory (cold), capacity, and conflict was first defined by Hill [30], and the stack algorithm for simulating a fully-associative cache is first described by Mattson et al. [44]. True and false sharing misses have been defined by Torrellas et al. [71], Eggers et al. [24], and Dubois et al. [23]. Each of them also describe an off-line classification algorithm. In contrast to these schemes and definitions, the mechanisms we describe are designed to be implemented in real hardware for integration with existing on-line performance debugging infrastructures.

Coherence Decoupling [33] speculatively reads data values from invalid cache lines to hide the latency of a cache miss caused by false sharing. It then uses the incoming (coherent) data values to verify successful value speculation. If the values differ, recovery action is triggered. The primary aim of this work was to hide latency of load instructions that read unchanged values. One of our design points (Zero OD, Zero AcT) in Figure 42 to classify coherence misses is similar to this implementation.

Numerous research proposals have been made for improving the performance counter infrastructure [59], attribution of performance-related events to particular instructions [19],

and for sampling and processing of profiling data [2, 46, 47, 84]. Our cache miss classification mechanisms are synergistic with improvements in performance counters, sampling, and profiling infrastructure. Our mechanisms provide on-line identification of specific types of cache misses, and this identification can be used to drive performance counters, attributed to particular instructions, and processed further to gain more insight into program behavior and performance. The better the profiling infrastructure, the more beneficial the results of our classification are to the programmer. Conversely, our scheme enhances the value of a profiling infrastructure by providing additional event types that can be profiled.

Our CMD scheme relies on an approximation of the LRU stack algorithm. A similar generational approach has been used by Kim et al. [38] to perform efficient cache simulations and, more recently, by Zhou et al. [80] to dynamically track working set sizes in order to help allocate pages of physical memory. Our generational scheme is designed for conflict miss detection in caches, so its state is updated and looked up much more often that the working set mechanism in [80]. Because of this consideration, we use Bloom filters and in-cache generation numbers to track members of each generation, rather than hardware-maintained link-lists. We believe that our Bloom-filter based approach could be used to simplify and speed up the implementation of Zhou et al.'s working set scheme.

Collins et al. [13] propose a hardware scheme, the CMT, that identifies conflict misses by storing the tag of the last evicted block from each set. On a cache miss, if the miss is to the most recent victim for the corresponding set, the miss is classified as a conflict miss. We expect this scheme to be less accurate than ours for at least two reasons. First, since the CMT only remembers the last evicted block from a set, when a large number of lines compete for a set, many of these misses will be incorrectly identified as capacity misses. Also, since the CMT does not track recency information across different sets, infrequently accessed sets may suffer capacity misses that are classified as conflict misses. It should be noted that the CMT is primarily directed towards helping prefetchers and victim caches, where performance improvement can be achieved with moderately accurate miss

classification. In contrast, our CMD scheme provides cache miss classification to drive performance counters and hardware-assisted profiling, where misclassification of capacity misses as conflict misses can mislead the programmer into fruitless data-padding optimizations.

5.6 Summary

Performance debugging of parallel applications is extremely challenging, but achieving good parallel performance is critical to justify the additional expense of parallel architectures. One particularly challenging performance debugging problem is determining the causes and sources of cache misses. This is especially true for parallel applications since they potentially suffer misses from sources not present for uniprocessor applications (e.g., false sharing and destructive sharing).

We propose schemes to perform on-the-fly classification of cache misses in hardware. Our scheme includes two primary mechanisms: a Conflict Miss Detector (CMD) that classifies replacement misses into capacity and conflict misses, and a False Sharing Detector (FSD) that classifies coherence misses into false and true sharing misses. We evaluate our scheme on SPLASH-2 and Parsec-1.0 benchmarks and find that it is provides a similar picture of cache miss-related performance problems to previously proposed off-line schemes. Combined with existing performance counter mechanisms or with more advanced profiling mechanisms, our scheme also pinpoints the code responsible for each type of cache misses in the application. Our work is inline with the thesis statement that low-cost and efficient hardware solutions can help programmers debug their programs to improve preformance.

CHAPTER VI

CONCLUSIONS

A successful software product should ensure both correctness and performance in order to satisfy end-users. Often times, programmers address correctness at the cost of degrading the program performance or break program correctness while trying to improve performance. Hence, ensuring both of these aspects successfully in a program is a great challenge facing software developers.

In this dissertation, we describe three innovative hardware solutions that are aimed at improving certain correctness and performance problems, namely, memory debugging, taint propagation and detection/classification of cache misses. Memory bugs are a frequent cause of program crashes and even security threats. MemTracker provides low-cost, efficient and programmable hardware support for memory access monitoring and debugging. Tainting is a popular dynamic information flow tracking mechanism frequently used in detecting malicious uses of values derived from " unsafe" inputs. FlexiTaint is our programmable hardware accelerator solution for dynamic taint propagation. With the growing popularity of multi-core machines, performance is becoming increasingly important and cache misses are notoriously known for being common performance bottlenecks. CacheDoc is a comprehensive cache miss classification tool implemented in hardware. It classifies replacement misses into capacity and conflict misses, and coherence misses into false and true sharing misses.

We adopt two common goals for all of our proposed solutions: i) Low cost, and ii) efficiency. Both of these goals are targeted at improving productivity of programmers and reducing software development costs. We also incorporate programmability as a key design choice in order to improve flexibility of hardware to implement checkers of users' choice

and also, to adapt to changing needs of programmers as new type of bugs are discovered.

Experiments show that our proposed tools incur low performance impact which enables their usage in live (production) runs. This is achieved by making minimal changes to the performance-critical parts of the processor pipeline. Implementation cost is kept low (in fact, negligible for some of our implementations for cache miss classification) by minimizing the amount of changes that are made to already existing processor hardware. We consider this dissertation as a step that would take us closer toward enhancing programmer's experience with low cost hardware designs.

Over the past years, innovations in process technology and architecture have paved the way to a host of high performance machines in the market. Better performance debugging tools are needed to effectively harness the potential unleashed by such powerful machines. Even our correctness debugging mechanisms are designed to take multi-core processors into consideration. While architects invest time and resources into designing high end architectures, we believe that it is equally important to incorporate useful debugging features into these processors in order to enhance the ease of use for programmers.

APPENDIX A

ATTRIBUTION OF CACHE MISSES

This appendix presents attribution results from experiments described in Chapter 5. To facilitate performance debugging efforts, it is important for programmers to know which set of static instructions (and from there, lines of source code in the program) suffer different types of cache misses. We measure the number of cache misses that are suffered by each static instruction. This helps capture the instructions that are involved in a large number of cache misses and software developers can then decide which code needs to be changed and what changes are needed.

Attribution results are shown for four different schemes, two for replacement miss classification and two for coherence miss classification. For classification of replacement misses (Tables 8–11), we show the top ten static instructions that suffer capacity and conflict misses in the off-line LRU stack and in the generational scheme with four generations for a 32 KB, 4-way private L1 cache. For coherence miss classification (Tables 12–15), we show top ten static instructions that suffer true and false sharing misses in the ideal I-FSD and in the (Partial(1k) OD, Partial(16k) AcT) configuration from Figure 42 for a 512 KB, 16-way private L2 cache. For each benchmark, we show the Program Counters (PCs) in the program that suffer the most misses and the percentage of replacement or coherence misses suffered by that PC.

Our results show that both the LRU stack and the generational scheme present identical results except for a few cases. In the barnes benchmark (Table 8), we observe that the static instruction at program counter `40677c` suffers a higher percentage of conflict misses than capacity misses according to the off-line I-CMD classification. However, our generational scheme reports a higher percentage of capacity misses than conflict misses on the same

instruction. On further investigation, we find that about 8% of replacement misses suffered by this static instruction occur in the bottom 25% of LRU stack (also referred to as "near-capacity" misses). These misses are misclassifed by our generational scheme due to loss of history when information about entire generation is erased. Another observation seen in many benchmarks is the minor shuffle in ranking of static instructions. In all these cases, we find that these static instructions are responsible for a similar ($<$1% difference) percentage of misses, so the programmer would still get the big picture (similar number of misses) even though the rank order is different.

For coherence miss classification, we find that the I-FSD and the (Partial(1k) OD, Partial(16k) AcT) show identical order for top five offending instructions except in a very few benchmarks. In cases where there is slight difference in ordering, we find that the corresponding instructions contribute $<$2% of all coherence misses. Overall, we find that partial state schemes capture very good accuracy and the ordering of offending instructions for all cases that contribute significantly ($>$2%) to coherence misses.

Table 8: Attribution of Replacement Misses. Capacity and conflict misses are shown as percentage of replacement misses.

LRU Stack				Generational CMD			
Capacity Misses		Conflict Misses		Capacity Misses		Conflict Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
barnes							
40677c	13.52	40677c	22.38	40677c	21.69	40677c	14.74
4065dc	7.47	4067b8	11.65	4067b8	11.19	4067b8	11.43
4067b8	6.51	406698	6.20	406698	6.35	406698	5.84
406698	6.35	4068f0	5.15	4068f0	4.93	4068f0	5.10
4065ec	2.96	406660	1.10	4065dc	1.47	406660	1.12
4068f0	2.92	406814	0.92	406660	1.15	406814	0.95
4067dc	1.88	4067dc	0.88	4067dc	0.98	4067dc	0.90
403e84	1.86	4065dc	0.78	406814	0.92	4065dc	0.79
403eec	1.78	4065ec	0.44	4065ec	0.70	418da0	0.73
406824	1.76	418da0	0.25	406824	0.32	418db8	0.61
cholesk							
40bd34	21.55	40ae68	11.77	40bd34	20.91	40ae68	9.62
40ae68	6.02	402ff4	8.15	40ae68	6.44	402ff4	5.73
406270	3.60	402fe8	5.45	402ff4	3.78	405d9c	4.01
402ff4	3.39	402ef4	3.27	406270	3.49	402ef4	3.46
405d9c	3.37	405d9c	2.76	405d9c	3.28	402fe8	3.28
402fe8	2.53	40ae58	2.06	402fe8	2.83	41ff20	2.30
40a7fc	2.34	40aec0	1.63	40a7fc	2.28	40ae58	1.69
40b5a0	2.24	41ff20	1.57	40b5a0	2.18	403068	1.55
40a7e8	2.00	40e848	1.43	40a7e8	1.96	40aec0	1.49
40a218	1.72	4028a0	1.39	40a218	1.67	41ff38	1.38
fft							
40084c	5.54	4138f0	6.63	40084c	5.54	4138f0	6.75
400738	5.54	413908	5.91	400738	5.54	413908	6.00
400854	5.51	413904	4.07	400854	5.50	413904	4.16
400740	5.50	40bde0	3.35	400740	5.49	40bde0	3.42
4028ec	3.76	401154	1.33	4028ec	3.75	401154	0.55
402bc0	3.75	4011f8	0.54	402bc0	3.75	4011f8	0.55
4014d8	3.74	401150	0.41	4014d8	3.74	40d4a8	0.42
413908	1.88	40d4a8	0.41	413908	1.90	410134	0.19
4011f8	1.80	410134	0.20	4011f8	1.80	435854	0.18
4138f0	1.52	435854	0.19	4138f0	1.53	410138	0.17
fmm							
40c688	16.98	40c688	12.89	40c688	17.73	40c688	10.02
40b328	9.70	40c744	9.08	40b328	9.92	40c744	9.96
40b330	4.73	40b328	6.33	40b330	4.87	40b328	4.71
40c744	2.40	40c7e0	3.92	40c744	3.47	420920	4.21
40b32c	2.34	420920	3.42	40b32c	2.39	40c7e0	4.20
40be1c	2.25	40b330	3.29	420938	1.95	420938	2.59
40b868	2.24	420938	2.12	40be1c	1.91	40b330	2.51
40b8e8	2.22	40c784	1.97	40b868	1.83	40c784	2.34
40b754	2.18	40b32c	1.76	40b8e8	1.82	402ef0	2.03
40c320	2.13	402ef0	1.51	40b754	1.81	402f88	1.87

Table 9: Attribution of Replacement Misses. Capacity and conflict misses are shown as percentage of replacement misses.

LRU Stack				Generational CMD			
Capacity Misses		Conflict Misses		Capacity Misses		Conflict Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
lu							
401784	15.30	40ef10	13.49	401784	14.78	40ef10	14.06
401764	9.14	40ef28	11.61	401764	8.55	40ef28	10.81
401780	9.01	40ef24	6.09	401780	8.35	40ef24	6.32
40ef28	3.29	407400	5.73	40ef28	4.45	407400	5.64
400ea8	2.70	401764	2.93	400ea8	2.48	401764	3.06
401008	2.65	401784	2.40	401008	2.44	401780	1.79
40ef10	1.75	401780	1.69	40ef10	2.34	401784	1.57
407400	0.93	40b728	0.28	407400	1.37	40b728	0.30
401b0c	0.60	40eb9c	0.19	40ef24	0.82	4073f0	0.18
40ef24	0.52	4073f0	0.19	401b0c	0.63	407c3c	0.18
ocean							
404d3c	9.72	426488	20.54	426488	12.27	426470	20.46
404cfc	7.87	426470	19.99	426470	7.88	426488	19.93
426488	5.28	41e960	6.85	404d3c	6.44	41e960	7.25
405b7c	4.61	405d44	0.41	404cfc	5.19	4264f0	0.34
405c48	4.60	405000	0.33	405b7c	3.33	426074	0.29
405d44	3.64	405b7c	0.32	405c48	3.28	405d44	0.26
405000	3.44	4264f0	0.32	41e960	3.04	422cb8	0.25
405b74	3.39	405c48	0.28	405d44	2.77	405000	0.23
405d3c	3.16	405b74	0.27	405000	2.53	41e950	0.23
405d34	3.11	426074	0.26	405b74	2.49	426424	0.22
radiosity							
40af68	20.08	430d1c	17.11	40af68	19.80	430d1c	17.19
407598	5.37	430cf4	12.16	407598	5.33	430cf4	11.95
430cf4	4.85	42a068	2.57	430cf4	5.14	42a068	2.41
40960c	3.35	41991c	2.29	40960c	3.30	41991c	2.10
41777c	2.74	41b504	1.57	41777c	2.70	41b504	1.60
407660	2.50	41b770	1.34	407660	2.49	41b770	1.36
42c4d8	2.34	41ce8c	1.11	42c4d8	2.33	41ce8c	1.10
418120	1.71	42c4d8	1.08	418120	1.69	42c4d8	1.06
419a4c	1.65	40ad24	0.92	419a4c	1.65	40ad24	0.93
407668	1.22	41cedc	0.91	430d1c	1.43	41cedc	0.91
radix							
4019a4	17.21	4019a4	10.57	4019a4	17.53	4019a4	9.79
401970	4.09	40ab40	5.77	401970	4.23	40ab40	5.68
40159c	4.00	401970	3.54	40159c	3.96	401970	3.36
400d10	3.20	40ab58	2.60	4010d8	3.12	401470	1.87
4010d8	3.12	403030	2.09	400d10	3.06	40ab58	1.86
40ab58	2.39	401470	1.88	40ab58	2.77	403030	1.27
401204	1.60	401854	1.46	401204	1.53	401854	0.95
401068	1.59	4012c8	0.91	401068	1.53	401990	0.83
401514	1.56	401990	0.88	401514	1.50	4012c8	0.81
401214	1.52	4012d0	0.55	403030	1.46	4012d0	0.50

Table 10: Attribution of Replacement Misses.Capacity and conflict misses are shown as percentage of replacement misses.

LRU Stack				Generational CMD			
Capacity Misses		Conflict Misses		Capacity Misses		Conflict Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
raytrace							
405504	7.91	42d178	23.03	405504	6.90	42d178	23.44
4056c8	6.98	42d160	21.76	42d178	6.59	42d160	22.28
402b0c	6.21	40abf8	0.85	4056c8	6.11	42d174	0.57
40abf8	5.52	40bc10	0.64	42d160	5.57	40abf8	0.53
40550c	3.98	402b0c	0.59	402b0c	5.51	40bc10	0.44
405518	3.97	4056c8	0.50	40abf8	5.04	402b0c	0.38
402ba8	3.88	405504	0.48	40550c	3.46	4056c8	0.32
40a92c	3.61	42d174	0.47	405518	3.45	40a92c	0.32
402b74	3.60	40a92c	0.44	402ba8	3.45	402ba8	0.30
4056d0	3.52	402ba8	0.43	40a92c	3.23	42805c	0.29
water-sp							
402ad4	7.36	4200c0	9.45	402ad4	7.31	4200c0	9.60
402ab4	6.34	4200d8	6.60	402ab4	6.26	4200d8	6.68
4039dc	4.74	413c38	5.06	4039dc	4.67	413c38	4.75
4200d8	4.67	413c34	4.21	4200d8	4.63	413c34	3.98
413c38	4.03	4185b0	3.68	413c38	4.31	4185b0	3.73
413c34	2.78	402ad4	3.32	413c34	2.99	402ad4	3.31
40362c	2.66	4200d4	1.61	40362c	2.63	4200d4	1.63
4200c0	2.04	402ae8	0.85	4200c0	2.02	402ae8	0.85
40708c	1.92	402af4	0.75	40708c	1.89	402ab4	0.76
4185b0	1.40	402ab4	0.75	4185b0	1.39	402af4	0.75
water-n2							
401a44	15.19	420010	12.40	401a44	15.11	420010	13.13
401a24	7.03	420028	7.15	401a24	6.98	420028	7.53
40318c	6.95	418500	4.41	40318c	6.91	418500	4.66
402fd0	5.71	401a44	3.27	402fd0	5.68	401a44	3.22
413b88	3.01	413b88	3.21	413b88	3.17	413b88	1.92
402fc4	2.94	413b84	2.38	402fc4	2.92	413b84	1.46
4376b4	2.32	402ae0	0.32	413b84	2.39	419bc8	0.33
413b84	2.28	419bc8	0.31	4376b4	2.31	401a24	0.30
406348	1.58	402a88	0.30	406348	1.57	402ae0	0.28
420028	0.96	401a24	0.29	420028	0.95	402a88	0.26
blkscholes							
402608	16.18	40cdb0	14.88	402608	15.98	40cdb0	14.83
400840	6.34	40cdc8	14.24	400840	6.26	40cdc8	13.70
400740	6.23	40cdc4	6.15	400740	6.16	40cdc4	6.31
40077c	5.76	4052a0	3.66	40077c	5.69	4052a0	3.79
407cac	3.09	4035ac	1.27	40cdc8	3.26	4035ac	0.97
40cdc8	2.80	407cac	0.59	407cac	3.08	407cac	0.56
40069c	1.92	40ca60	0.31	40069c	1.90	40ca60	0.29
400694	1.88	40c9b4	0.26	400694	1.86	40c9b4	0.26
4008a4	1.75	40ce3c	0.21	4008a4	1.73	400840	0.22
400ac0	1.37	400840	0.21	40cdb0	1.44	40ce3c	0.22

Table 11: Attribution of Replacement Misses. Capacity and conflict misses are shown as percentage of replacement misses.

LRU Stack				Generational CMD			
Capacity Misses		Conflict Misses		Capacity Misses		Conflict Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
bodytrack							
400b20	9.55	400b64	5.48	400b20	9.42	400b64	5.86
42f104	8.71	41b8a8	5.41	42f104	8.57	41b8a4	5.74
400b64	5.73	41b8a4	5.37	400b64	5.64	4037cc	5.16
400688	5.39	4037cc	5.34	400688	5.29	41b8a8	4.77
400820	4.36	41b890	4.43	400820	4.28	400b20	4.14
41b8a8	3.39	400b20	4.02	41b8a8	3.65	41b890	4.14
402318	2.69	4037ac	3.73	41b890	2.79	4037ac	3.64
41b890	2.65	4010ec	3.60	4037ac	2.71	4010ec	3.45
4037ac	2.64	413d80	2.51	402318	2.67	413d80	2.10
40242c	2.11	4004d4	1.53	40242c	2.25	4004d4	1.66
facesim							
401a34	22.55	40c190	14.21	401a34	22.21	40c190	14.36
401538	12.17	40c1a8	14.19	401538	12.03	40c1a8	14.28
40708c	2.45	40c1a4	6.38	40708c	2.50	40c1a4	6.46
401418	2.32	404680	2.59	401418	2.40	404680	2.54
40c1a8	1.54	401418	1.19	40c1a8	1.73	401418	1.00
4013c8	1.30	40708c	1.00	4013c8	1.33	40708c	0.87
40c190	0.97	4013c8	0.92	40c190	1.15	4013c8	0.85
406cc0	0.96	401538	0.38	406cc0	0.94	401538	0.31
40be40	0.70	40298c	0.34	40be40	0.72	40be40	0.29
40153c	0.61	40be40	0.34	40153c	0.61	40c21c	0.27
fluidnanim							
404390	5.76	40fac8	17.65	40fac8	6.23	40fac8	17.03
404414	5.74	40fab0	16.29	404390	5.48	40fab0	16.18
40452c	5.72	40fac4	6.79	404414	5.45	40fac4	7.31
404420	5.16	404680	2.49	40452c	5.44	404680	2.65
404648	4.97	4062ac	2.36	404420	4.91	4062ac	1.55
40fac8	4.77	407fa0	0.85	404648	4.75	407fa0	0.87
40446c	4.52	40a9ac	0.58	40fab0	4.40	404648	0.61
4043b0	4.52	404648	0.55	40446c	4.29	40fb3c	0.61
404470	4.52	40fb3c	0.50	404470	4.29	40f6b4	0.54
404680	3.84	40f6b4	0.42	4043b0	4.29	40a9ac	0.50
swaptions							
400e52	21.01	400c24	13.56	400e52	21.01	400c24	13.42
400c24	21.01	400e52	13.35	400c24	21.01	400e52	13.27
400c3c	1.52	409170	8.32	400024	1.52	409170	8.49
400024	1.51	400d28	1.23	400c3c	1.52	400b4c	1.21
400c30	1.40	400b4c	1.22	400c30	1.40	400d28	1.21
403800	1.39	400bac	0.91	400082	1.39	400bac	0.90
409188	0.72	400d88	0.90	409188	0.72	400d88	0.88
4008a8	0.67	409188	0.59	4008a8	0.67	409188	0.61
401660	0.32	400b70	0.56	401660	0.32	400b70	0.52
409170	0.18	400d44	0.54	409170	0.18	400d44	0.49

Table 12: Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses.

I-FSD				(Partial(1k) OD, Partial(16k) AcT)			
TrueSharing Misses		FalseSharing Misses		TrueSharing Misses		FalseSharing Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
Barnes							
4141cc	9.65	413c9c	11.96	4141cc	9.50	413c9c	12.22
418b34	4.51	4141cc	1.90	418b34	4.44	4141cc	1.94
418a50	3.06	4065ec	1.31	4065dc	3.07	408464	1.28
4065dc	3.00	408464	1.26	418a50	3.02	4065ec	1.27
406698	2.63	406698	1.14	406698	3.01	4065dc	0.74
418d10	2.45	4065dc	0.95	418d10	2.63	408708	0.67
418d20	1.48	408708	0.66	418d20	1.46	40848c	0.55
4065ec	1.19	40848c	0.55	4065ec	1.18	406698	0.52
413c9c	1.12	418d10	0.33	413c9c	1.10	408634	0.28
418a2c	0.97	408634	0.28	418a2c	0.96	418b34	0.23
cholesky							
41fbd0	7.33	40c2fc	29.97	41fbd0	4.88	40c2fc	30.48
40c2f0	2.64	41ae1c	19.15	40c2d4	3.55	41ae1c	19.32
40c530	2.16	40bdfc	8.34	40c2f0	1.82	40bdfc	8.52
40c2d4	1.99	40c29c	5.01	40c530	1.44	40c29c	5.11
41fea0	1.73	40c2e4	3.87	41fea0	1.15	40c2e4	3.95
41ae1c	1.31	40c2d4	3.50	41ae1c	1.03	40aa7c	2.69
40c2fc	1.08	40aa7c	2.66	40c2fc	0.81	41b34c	2.14
40c494	0.93	41b34c	2.11	40c494	0.62	40bc24	0.39
40aa7c	0.66	40bc24	0.39	40aa7c	0.46	40adf0	0.34
40c39c	0.65	40c2f0	0.36	40c39c	0.43	40c39c	0.33
fft							
40ed1c	19.96	40d1a8	33.93	40ed1c	19.92	40d1a8	35.19
40e7ec	15.61	40e7ec	9.52	40e7ec	15.58	40e7ec	9.88
4135a0	9.19	40be50	1.79	4135a0	9.17	40fc80	0.62
413224	1.88	40bd78	1.19	413224	1.87	40beb0	0.62
42a87c	0.79	40fc80	0.60	42a87c	0.79	40d498	0.62
413684	0.59	40beb0	0.60	413684	0.59	41309c	0.62
41320c	0.49	40d498	0.60	40be50	0.59	40cd20	0.62
41321c	0.49	41309c	0.60	41320c	0.49	413908	0.62
42a60c	0.49	40cd20	0.60	41321c	0.49	4109a4	0.62
40c618	0.49	413908	0.60	42a60c	0.49	40bdc8	0.62
fmm							
4205d0	10.55	41b81c	14.34	4205d0	10.84	41b81c	14.67
41bd4c	10.04	40c5d4	7.38	41bd4c	10.30	40c5d4	7.41
4206b4	6.27	41bd4c	3.29	4206b4	6.44	41bd4c	3.37
4205ac	3.17	40c93c	2.93	4205ac	3.25	40c93c	3.00
4208a0	3.03	4206c0	1.48	4208a0	3.11	4206c0	1.43
409c7c	2.56	418e84	1.26	409c7c	2.62	418ecc	1.01
40c064	2.54	418ecc	1.10	4206c8	1.96	41a1d8	0.88
4206c8	1.91	41a1d8	0.86	420524	1.62	418ee0	0.72
420524	1.58	418ee0	0.78	40c014	1.38	40ba7c	0.72
40c5d4	1.43	40ba7c	0.70	42059c	1.37	40bdd0	0.52

Table 13: Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses.

I-FSD				(Partial(1k) OD, Partial(16k) AcT)			
TrueSharing Misses		FalseSharing Misses		TrueSharing Misses		FalseSharing Misses	
TrueSh	Misses	FalseSh	Misses	TrueSh	Misses	FalseSh	Misses
PC	Percent	PC	Percent	PC	Percent	PC	Percent
lu							
40ebc0	18.25	40ecb0	21.30	40ebc0	17.82	40ecb0	22.32
40a33c	11.38	409e0c	9.71	40a33c	11.11	409e0c	10.34
40ecb0	5.54	407470	2.45	40ecb0	5.56	402194	0.80
40e83c	3.20	402194	0.75	40e83c	3.12	4087c8	0.79
40eb14	1.30	4087c8	0.74	407470	1.52	40ef28	0.63
40eca4	1.23	40ef28	0.59	40eb14	1.26	4074bc	0.41
40eb9c	0.95	4074bc	0.38	40eca4	1.20	40a33c	0.27
409e0c	0.76	407398	0.37	40eb9c	0.92	4074d0	0.25
40ee90	0.61	4074d0	0.25	409e0c	0.74	409268	0.10
407c38	0.47	40a33c	0.25	40ee90	0.60	40ebc0	0.04
ocean							
426120	6.49	42136c	9.94	426120	6.51	42136c	11.87
42189c	6.32	406ca4	9.80	42189c	6.33	406ca4	9.87
4073b4	6.19	406c48	5.55	4073b4	6.14	406c48	5.85
407600	5.85	41e9d0	3.70	407600	5.81	4075f4	1.45
40727c	5.24	41e8f8	2.98	40727c	5.19	407340	1.06
407350	4.60	4075f4	1.24	407350	4.56	42189c	0.98
407598	3.94	407340	0.91	407598	3.92	407268	0.97
4074c4	3.11	41ea34	0.88	4074c4	3.10	426488	0.88
426204	2.97	42189c	0.82	406ca4	2.99	4074ac	0.68
406ca4	2.81	407268	0.81	426204	2.98	407588	0.42
radiosity							
418c04	10.85	42c4d8	11.34	418c04	10.95	42c4d8	11.52
41991c	6.89	4190f8	10.15	41991c	6.86	4190f8	9.95
40ac78	4.59	40a024	7.51	40ac78	4.57	40a024	7.50
4309b0	4.00	41e634	1.41	4309b0	3.99	41e634	1.40
42c4d8	2.87	41913c	1.29	42c4d8	2.85	41913c	1.18
418a1c	2.75	40ae40	0.85	418a1c	2.78	40ae40	0.86
430a84	1.74	41991c	0.76	430a84	1.74	41991c	0.77
430c44	1.31	42c944	0.68	430c44	1.31	42c944	0.70
40ae34	1.23	418c04	0.63	40ae34	1.23	407598	0.55
42c944	1.20	407598	0.55	42c944	1.19	40aa98	0.51
radix							
4019a4	42.95	405a3c	4.96	4019a4	47.92	405a3c	5.07
40a7f0	1.71	400ddc	2.26	40a7f0	5.32	400ddc	2.31
40a8d4	1.33	4019a4	2.14	40a8d4	4.16	4019a4	2.12
4099b4	0.23	403100	1.04	4099b4	0.71	403100	0.83
4010d8	0.15	40113c	0.63	4030a0	0.57	40113c	0.57
405378	0.14	4030a0	0.59	4010d8	0.49	4099b4	0.38
4030a0	0.14	4030ec	0.52	405378	0.43	4030ec	0.30
40a744	0.13	4099b4	0.37	40a744	0.41	405f6c	0.23
40a464	0.12	4011f8	0.30	403100	0.40	4011f8	0.19
40aac0	0.12	405f6c	0.22	40a464	0.39	403020	0.15

Table 14: Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses.

I-FSD				(Partial(1k) OD, Partial(16k) AcT)			
TrueSharing Misses		FalseSharing Misses		TrueSharing Misses		FalseSharing Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
raytrace							
42858c	12.14	40ab8c	22.00	42858c	32.14	40ab8c	22.01
40ff6c	7.24	42805c	4.16	40ff6c	27.24	42805c	4.16
42d0d0	5.31	42858c	2.62	42d0d0	25.32	42858c	2.61
42cd64	3.52	40a894	1.97	42cd64	23.52	40a894	1.97
42ce10	3.08	40e794	0.59	42ce10	23.08	40e794	0.59
41057c	2.03	41145c	0.29	41057c	22.03	41145c	0.29
410100	1.90	411420	0.18	410100	21.90	411420	0.18
42d0e0	1.84	40c880	0.11	42d0e0	21.83	40c880	0.11
42cef4	1.33	426a18	0.10	42cef4	21.33	426a18	0.10
42d174	1.24	40d0bc	0.08	42d174	21.24	40d0bc	0.08
water-sp							
41fd70	10.61	41afbc	6.56	41fd70	10.58	41afbc	6.77
41b4ec	6.36	41b4ec	4.39	41b4ec	6.34	41b4ec	4.53
402ad4	5.77	418620	1.64	402ad4	5.76	400f58	1.66
402ab4	4.49	400f58	1.61	402ab4	4.48	41fe60	1.50
41fe54	3.77	41fe60	1.52	41fe54	3.76	418500	1.15
4039dc	2.55	418500	1.11	4039dc	2.54	404de4	0.86
40362c	2.04	404de4	0.84	40362c	2.03	419978	0.73
41fcc4	1.30	419978	0.71	41fcc4	1.30	41866c	0.38
41fe68	1.02	41866c	0.37	418620	1.09	4200d8	0.26
41f9f4	0.95	418548	0.31	41fe68	1.03	418680	0.22
water-n2							
41fcc0	15.20	41b43c	25.18	41fcc0	14.92	41b43c	25.82
41ff90	5.87	41af0c	5.04	41ff90	5.75	41af0c	5.16
41b43c	4.39	418570	1.04	41b43c	4.30	40e003	0.83
41ff78	2.50	40e003	0.81	41ff78	2.44	4198c8	0.75
41fda4	1.61	4198c8	0.73	41fda4	1.58	41a368	0.10
41fc14	1.44	418498	0.28	418570	1.56	4185bc	0.08
41af0c	0.82	41a368	0.10	41fc14	1.41	418450	0.08
41f92c	0.80	4185bc	0.08	41af0c	0.80	420028	0.08
418570	0.57	4185d0	0.08	41f92c	0.78	4185d0	0.07
41fdb8	0.51	418450	0.08	41fdb8	0.50	4184f0	0.02
blkscholes							
4008ac	10.43	400668	6.97	4008ac	10.67	400668	6.30
400658	6.55	400658	4.46	4035ac	6.58	400658	4.12
4035ac	6.46	40075c	2.65	400658	6.03	40075c	2.69
40ca60	3.67	4009f8	1.96	40ca60	3.75	407cac	1.91
4035fc	2.23	407cac	1.81	4035fc	2.29	4009f8	1.90
400a58	2.09	40cb50	1.72	400a58	2.03	40cb50	1.80
4009f8	1.99	400aa0	1.44	4009f8	1.84	400aa0	1.29
40075c	1.90	40085c	1.27	40075c	1.81	40085c	1.19
4081dc	1.42	4081dc	0.95	4081dc	1.44	4081dc	1.00
403678	1.39	400598	0.88	403678	1.42	40171c	0.55

Table 15: Attribution of Coherence Misses. True and False Sharing misses are shown as percentage of coherence misses.

I-FSD				(Partial(1k) OD, Partial(16k) AcT)			
TrueSharing Misses		FalseSharing Misses		TrueSharing Misses		FalseSharing Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
bodytrack							
400be4	15.79	400b64	19.73	400be4	15.80	400b64	19.82
400b20	7.19	400b20	10.79	400b20	5.98	400b20	10.77
402318	4.41	400b88	5.23	402318	4.71	400b88	5.13
400b64	4.00	400be4	1.54	400b64	3.76	400be4	1.50
400b88	3.43	41b630	1.01	400b88	3.52	41b630	1.02
41b540	2.43	41678c	0.37	41b540	2.59	41678c	0.38
40242c	1.94	413df0	0.08	40242c	2.08	415148	0.06
42f104	1.09	415148	0.06	42f104	1.14	416cbc	0.05
41b810	0.68	416cbc	0.05	41b800	0.73	413e3c	0.04
41b800	0.67	413e3c	0.04	41b810	0.73	4024c8	0.02
facesim							
40be40	7.31	40708c	17.41	40be40	8.66	40708c	17.62
40c110	4.14	4075bc	2.91	40c110	4.89	4075bc	2.94
401438	2.55	40bf30	2.10	401438	3.10	40bf30	2.12
40298c	2.42	401438	0.68	40298c	2.85	401438	0.62
401418	1.11	4046f0	0.43	401418	1.32	401418	0.38
40bf24	1.08	4014a4	0.41	40bf24	1.28	4014a4	0.37
40c0f8	1.06	401120	0.40	40c0f8	1.25	401120	0.24
4014a4	0.89	401418	0.38	40708c	0.87	401540	0.21
40708c	0.73	401540	0.23	4075bc	0.81	401100	0.18
4075bc	0.68	401100	0.19	4027f0	0.60	405a48	0.14
fluidnanim							
4062ac	17.32	40a9ac	14.37	4062ac	17.31	40a9ac	14.84
4072a0	9.50	40f850	7.43	4072a0	9.50	40f850	7.63
40f760	8.00	40805c	3.22	40f760	8.01	40805c	3.32
4062fc	5.47	408010	2.06	4062fc	5.48	404510	1.35
406378	1.93	404510	1.33	406378	1.93	40461c	0.94
40f844	1.53	40461c	0.93	40f844	1.53	404668	0.91
40fa30	1.51	404668	0.89	40fa30	1.50	40aedc	0.82
4072c4	1.40	40aedc	0.79	4072c4	1.40	40634c	0.64
4063d4	0.78	40634c	0.62	4063d4	0.78	40f3e4	0.61
4063c8	0.78	40f3e4	0.59	4063c8	0.77	407250	0.58
swaptions							
408e24	12.05	405094	6.27	408e22	11.92	405094	6.46
408a94	3.70	40406c	4.62	408a94	3.66	40406c	4.77
408d74	3.06	401734	0.88	408d74	3.02	408ed4	0.57
400cd4	2.27	408ed4	0.55	400cd4	2.25	402a28	0.42
400f64	2.25	402a28	0.41	400f64	2.22	40459c	0.11
408f04	1.88	40174c	0.39	408f04	1.86	4034c8	0.06
4090f0	0.89	4016d0	0.31	4090f0	0.88	409600	0.02
408914	0.77	40459c	0.11	401734	0.88	406224	0.02
401734	0.34	4034c8	0.06	408914	0.77	408e22	0.02
40406c	0.28	40171c	0.05	4016d0	0.30	4015b0	0.02

APPENDIX B

CACHE MISSES FOR DIFFERENT INPUTS

In this appendix, we perform experiments to verify if the cache misses occur in the same places in the program across different inputs. This is important because, if cache misses that occur frequently in one input set may not occur with different inputs, fixes applied based on results from one input set could degrade the program runtime when run with different input set. Therefore, it is necessary to verify that different inputs result in similar program behavior with respect to cache misses suffered by various static instructions.

We use Splash-2 benchmarks with inputs shown in Table 16 and Parsec-1.0 benchmarks with simlarge and simmedium input sets. We present the top ten offending instructions (PC) that cause conflict misses and false sharing misses and check whether the instructions match between two input sets. For conflict misses, we observe that the top offending instructions mostly match except in a few cases. For example, in ocean benchmark (Table 18), conflict misses caused by static instructions at addresses `426488` and `426470` contribute to about 40% of replacement misses in simlarge, but they account only for 12.5% on simmedium inputs. Consequently, a profiling run using a smaller input may not lead the programmer to fix the capacity miss problem that occurs with larger inputs. We observe a similar behavior for false sharing misses in simlarge and simmedium inputs. For example, in lu benchmark (Table 18), false sharing misses caused by static instructions at addresses `40ecb0` and `409e0c` contribute to about 30% of coherence misses on the simlarge input whereas, they account for roughly 12.75% on the simmedium input. However, in general, we observe that the simlarge and simmedium inputs have similar trend in conflict and false sharing misses in most benchmarks, and that runs with different inputs tend to have the same static instructions as primary culprits for each kind of cache misses, even though runs with smaller

Table 16: Splash-2 Benchmarks with different inputs.

Benchmark	Simlarge Input	Simmedium Input
Barnes	16K	8K
Cholesky	tk29.0	tk23.O
FFT	64K	32K
FMM	16K	8K
LU	512x512	256x256
Ocean	258x258	130x130
Radiosity	-room	-batch
Radix	256K	128K
Raytrace	car	teapot
Water-sp	512	256
Water-n2	512	256

inputs underestimate the importance of top offending static instructions relative to lower-ranked instructions.

Table 17: Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses.

Simlarge input				Simmedium input			
Conflict Misses		FalseSharing Misses		Conflict Misses		FalseSharing Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
barnes							
40677c	22.38	413c9c	11.96	40677c	21.73	413c9c	17.49
4067b8	11.65	4141cc	1.90	4067b8	12.30	4141cc	12.49
406698	6.20	4065ec	1.31	406698	6.36	408464	1.95
4068f0	5.15	408464	1.26	4068f0	5.65	4065ec	1.24
406660	1.10	406698	1.14	406660	0.98	4065dc	1.08
406814	0.92	4065dc	0.95	406814	0.74	408708	0.76
4067dc	0.88	408708	0.66	4067dc	0.67	40848c	0.76
4065dc	0.78	40848c	0.55	4065dc	0.65	406698	0.74
4065ec	0.44	418d10	0.33	418da0	0.39	408634	0.70
418da0	0.25	408634	0.28	418db8	0.13	418b34	0.66
cholesky							
40ae68	11.77	40c2fc	29.97	40ae68	6.66	40c2fc	22.05
402ff4	8.15	41ae1c	19.15	402ff4	6.38	41ae1c	12.85
402fe8	5.45	40bdfc	8.34	405d9c	3.07	40bdfc	4.20
402ef4	3.27	40c29c	5.01	402ef4	2.92	40c29c	3.62
405d9c	2.76	40c2e4	3.87	402fe8	2.44	40c2e4	2.37
40ae58	2.06	40c2d4	3.50	41ff20	2.19	40aa7c	2.08
40aec0	1.63	40aa7c	2.66	40ae58	1.84	41b34c	1.42
41ff20	1.57	41b34c	2.11	403068	1.53	40bc24	0.57
40e848	1.43	40bc24	0.39	40aec0	1.52	40adf0	0.45
4028a0	1.39	40c2f0	0.36	41ff38	1.42	40c39c	0.39
fft							
4138f0	6.63	40d1a8	33.93	4138f0	3.28	40d1a8	19.54
413908	5.91	40e7ec	9.52	413908	1.07	40e7ec	18.20
413904	4.07	40be50	1.79	413904	0.19	40fc80	6.90
40bde0	3.35	40bd78	1.19	40bde0	0.13	40beb0	2.49
401154	1.33	40fc80	0.60	401154	0.10	40d498	1.53
4011f8	0.54	40beb0	0.60	4011f8	0.09	41309c	0.96
401150	0.41	40d498	0.60	40d4a8	0.09	40cd20	0.19
40d4a8	0.41	41309c	0.60	410134	0.09	413908	0.19
410134	0.20	40cd20	0.60	435854	0.08	4109a4	0.00
435854	0.19	413908	0.60	410138	0.07	40bdc8	0.00
fmm							
40c688	12.89	41b81c	14.34	40c688	17.19	41b81c	13.94
40c744	9.08	40c5d4	7.38	40c744	6.97	40c5d4	11.34
40b328	6.33	41bd4c	3.29	40b328	3.88	41bd4c	4.93
40c7e0	3.92	40c93c	2.93	420920	2.67	40c93c	4.42
420920	3.42	4206c0	1.48	40c7e0	2.48	4206c0	3.91
40b330	3.29	418e84	1.26	420938	1.93	418ecc	2.61
420938	2.12	418ecc	1.10	40b330	1.89	41a1d8	1.89
40c784	1.97	41a1d8	0.86	40c784	1.83	418ee0	1.12
40b32c	1.76	418ee0	0.78	402ef0	1.83	40ba7c	1.06
402ef0	1.51	40ba7c	0.70	402f88	1.52	40bdd0	0.29

Table 18: Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses.

Simlarge input				Simmedium input			
Conflict Misses		FalseSharing Misses		Conflict Misses		FalseSharing Misses	
Conflict	Misses	FalseSh	Misses	Conflict	Misses	FalseSh	Misses
PC	Percent	PC	Percent	PC	Percent	PC	Percent
lu							
40ef10	13.49	40ecb0	21.30	40ef10	13.44	40ecb0	6.79
40ef28	11.61	409e0c	9.71	40ef28	10.55	409e0c	5.95
40ef24	6.09	407470	2.45	40ef24	2.41	402194	4.56
407400	5.73	402194	0.75	407400	0.77	4087c8	0.73
401764	2.93	4087c8	0.74	401764	0.77	40ef28	0.43
401784	2.40	40ef28	0.59	401780	0.62	4074bc	0.32
401780	1.69	4074bc	0.38	401784	0.22	40a33c	0.20
40b728	0.28	407398	0.37	40b728	0.21	4074d0	0.18
40eb9c	0.19	4074d0	0.25	4073f0	0.17	409268	0.13
4073f0	0.19	40a33c	0.25	407c3c	0.13	40ebc0	0.10
ocean							
426488	20.54	42136c	9.94	426470	8.77	42136c	10.97
426470	19.99	406ca4	9.80	426488	4.93	406ca4	6.30
41e960	6.85	406c48	5.55	41e960	4.65	406c48	5.59
405d44	0.41	41e9d0	3.70	4264f0	4.19	4075f4	3.73
405000	0.33	41e8f8	2.98	426074	3.54	407340	0.94
405b7c	0.32	4075f4	1.24	405d44	3.36	42189c	0.54
4264f0	0.32	407340	0.91	422cb8	3.20	407268	0.46
405c48	0.28	41ea34	0.88	405000	2.98	426488	0.38
405b74	0.27	42189c	0.82	41e950	2.75	4074ac	0.36
426074	0.26	407268	0.81	426424	2.24	407588	0.35
radiosity							
430d1c	17.11	42c4d8	11.34	430d1c	20.61	42c4d8	14.22
430cf4	12.16	4190f8	10.15	430cf4	10.83	4190f8	11.30
42a068	2.57	40a024	7.51	42a068	3.04	40a024	8.68
41991c	2.29	41e634	1.41	41991c	1.97	41e634	1.67
41b504	1.57	41913c	1.29	41b504	1.54	41913c	1.23
41b770	1.34	40ae40	0.85	41b770	1.07	40ae40	1.23
41ce8c	1.11	41991c	0.76	41ce8c	0.57	41991c	1.18
42c4d8	1.08	42c944	0.68	42c4d8	0.46	42c944	0.91
40ad24	0.92	418c04	0.63	40ad24	0.40	407598	0.85
41cedc	0.91	407598	0.55	41cedc	0.18	40aa98	0.71
radix							
4019a4	10.57	405a3c	4.96	4019a4	8.57	405a3c	6.37
40ab40	5.77	400ddc	2.26	40ab40	5.99	400ddc	5.57
401970	3.54	4019a4	2.14	401970	5.42	4019a4	2.09
40ab58	2.60	403100	1.04	401470	5.13	403100	1.35
403030	2.09	40113c	0.63	40ab58	3.92	40113c	0.84
401470	1.88	4030a0	0.59	403030	3.65	4099b4	0.45
401854	1.46	4030ec	0.52	401854	3.60	4030ec	0.35
4012c8	0.91	4099b4	0.37	401990	3.29	405f6c	0.29
401990	0.88	4011f8	0.30	4012c8	3.28	4011f8	0.26
4012d0	0.55	405f6c	0.22	4012d0	3.10	403020	0.26

Table 19: Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses.

Simlarge input				Simmedium input			
Conflict Misses		FalseSharing Misses		Conflict Misses		FalseSharing Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
raytrace							
42d178	23.03	40ab8c	22.00	42d178	10.51	40ab8c	18.90
42d160	21.76	42805c	4.16	42d160	8.80	42805c	17.52
40abf8	0.85	42858c	2.62	42d174	7.50	42858c	3.22
40bc10	0.64	40a894	1.97	40abf8	5.17	40a894	3.09
402b0c	0.59	40e794	0.59	40bc10	5.10	40e794	0.96
4056c8	0.50	41145c	0.29	402b0c	4.30	41145c	0.36
405504	0.48	411420	0.18	4056c8	3.51	411420	0.17
42d174	0.47	40c880	0.11	40a92c	3.02	40c880	0.16
40a92c	0.44	426a18	0.10	402ba8	1.09	426a18	0.09
402ba8	0.43	40d0bc	0.08	42805c	1.01	40d0bc	0.05
water-sp							
4200c0	9.45	41afbc	6.56	4200c0	11.49	41afbc	6.32
4200d8	6.60	41b4ec	4.39	4200d8	10.69	41b4ec	1.99
413c38	5.06	418620	1.64	413c38	7.46	400f58	1.65
413c34	4.21	400f58	1.61	413c34	3.55	41fe60	1.55
4185b0	3.68	41fe60	1.52	4185b0	3.14	418500	1.04
402ad4	3.32	418500	1.11	402ad4	3.10	404de4	0.90
4200d4	1.61	404de4	0.84	4200d4	3.06	419978	0.64
402ae8	0.85	419978	0.71	402ae8	2.62	41866c	0.50
402af4	0.75	41866c	0.37	402ab4	2.46	4200d8	0.50
402ab4	0.75	418548	0.31	402af4	2.44	418680	0.37
water-n2							
420010	12.40	41b43c	25.18	420010	13.00	41b43c	22.70
420028	7.15	41af0c	5.04	420028	8.61	41af0c	14.75
418500	4.41	418570	1.04	418500	4.81	40e003	9.61
401a44	3.27	40e003	0.81	401a44	0.51	4198c8	1.00
413b88	3.21	4198c8	0.73	413b88	0.32	41a368	0.60
413b84	2.38	418498	0.28	413b84	0.23	4185bc	0.47
402ae0	0.32	41a368	0.10	419bc8	0.14	418450	0.33
419bc8	0.31	4185bc	0.08	401a24	0.12	420028	0.27
402a88	0.30	4185d0	0.08	402ae0	0.10	4185d0	0.20
401a24	0.29	418450	0.08	402a88	0.09	4184f0	0.07
blkscholes							
40cdb0	14.88	400668	6.97	40cdb0	14.17	400668	5.96
40cdc8	14.24	400658	4.46	40cdc8	9.77	400658	5.77
40cdc4	6.15	40075c	2.65	40cdc4	3.74	40075c	4.13
4052a0	3.66	4009f8	1.96	4052a0	1.68	407cac	0.80
4035ac	1.27	407cac	1.81	4035ac	1.18	4009f8	0.59
407cac	0.59	40cb50	1.72	407cac	0.93	40cb50	0.48
40ca60	0.31	400aa0	1.44	40ca60	0.77	400aa0	0.39
40c9b4	0.26	40085c	1.27	40c9b4	0.71	40085c	0.37
40ce3c	0.21	4081dc	0.95	400840	0.41	4081dc	0.23
400840	0.21	400598	0.88	40ce3c	0.34	40171c	0.09

Table 20: Attribution of Conflict and False Sharing Misses for different inputs. Conflict misses are shown as percentage of replacement misses and False Sharing Misses are shown as percentage of coherence misses.

Simlarge input				Simmedium input			
Conflict Misses		FalseSharing Misses		Conflict Misses		FalseSharing Misses	
PC	Percent	PC	Percent	PC	Percent	PC	Percent
bodytrack							
400b64	5.48	400b64	19.73	400b64	6.93	400b64	17.60
41b8a8	5.41	400b20	10.79	41b8a4	5.23	400b20	14.22
41b8a4	5.37	400b88	5.23	4037cc	4.67	400b88	4.43
4037cc	5.34	400be4	1.54	41b8a8	4.51	400be4	3.51
41b890	4.43	41b630	1.01	400b20	4.30	41b630	2.22
400b20	4.02	41678c	0.37	41b890	4.07	41678c	1.34
4037ac	3.73	413df0	0.08	4037ac	3.15	415148	0.88
4010ec	3.60	415148	0.06	4010ec	2.46	416cbc	0.86
413d80	2.51	416cbc	0.05	413d80	1.98	413e3c	0.65
4004d4	1.53	413e3c	0.04	4004d4	1.20	4024c8	0.57
facesim							
40c190	14.21	40708c	17.41	40c190	12.99	40708c	19.21
40c1a8	14.19	4075bc	2.91	40c1a8	11.05	4075bc	11.33
40c1a4	6.38	40bf30	2.10	40c1a4	7.98	40bf30	6.19
404680	2.59	401438	0.68	404680	3.44	401438	1.87
401418	1.19	4046f0	0.43	401418	2.99	401418	1.36
40708c	1.00	4014a4	0.41	40708c	1.77	4014a4	0.63
4013c8	0.92	401120	0.40	4013c8	1.20	401120	0.09
401538	0.38	401418	0.38	401538	1.10	401540	0.07
40298c	0.34	401540	0.23	40be40	0.50	401100	0.07
40be40	0.34	401100	0.19	40c21c	0.39	405a48	0.03
fluidanim							
40fac8	17.65	40a9ac	14.37	40fac8	18.92	40a9ac	14.78
40fab0	16.29	40f850	7.43	40fab0	13.88	40f850	13.91
40fac4	6.79	40805c	3.22	40fac4	8.74	40805c	4.87
404680	2.49	408010	2.06	404680	2.00	404510	1.70
4062ac	2.36	404510	1.33	4062ac	1.94	40461c	1.49
407fa0	0.85	40461c	0.93	407fa0	1.77	404668	1.45
40a9ac	0.58	404668	0.89	404648	0.49	40aedc	0.69
404648	0.55	40aedc	0.79	40fb3c	0.43	40634c	0.68
40fb3c	0.50	40634c	0.62	40f6b4	0.42	40f3e4	0.47
40f6b4	0.42	40f3e4	0.59	40a9ac	0.26	407250	0.41
swaptions							
400c24	13.56	405094	6.27	400c24	13.01	405094	6.40
400e52	13.35	40406c	4.62	400e52	10.72	40406c	3.47
409170	8.32	401734	0.88	409170	0.04	408ed4	0.72
400d28	1.23	408ed4	0.55	400b4c	0.04	402a28	0.12
400b4c	1.22	402a28	0.41	400d28	0.03	40459c	0.06
400bac	0.91	40174c	0.39	400bac	0.03	4034c8	0.06
400d88	0.90	4016d0	0.31	400d88	0.02	409600	0.00
409188	0.59	40459c	0.11	409188	0.02	406224	0.00
400b70	0.56	4034c8	0.06	400b70	0.02	408e22	0.00
400d44	0.54	40171c	0.05	400d44	0.02	4015b0	0.00

REFERENCES

[1] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., and CASTRO, M., "Preventing Memory Error Exploits with WIT," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*, (Washington, DC, USA), pp. 263–277, IEEE Computer Society, 2008.

[2] ALEXANDROV, A., BRATANOV, S., FEDOROVA, J., LEVINTHAL, D., LOPATIN, I., and RYABTSEV, D., "Parallelization Made Easier with Intel Performance-Tuning Utility," *Intel Technology Journal*, Nov. 2007.

[3] BIENIA, C., KUMAR, S., SINGH, J., and LI, K., "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Princeton University Technical Report TR-811-08*, Jan. 2008.

[4] BLOOM, B. H., "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM 13(7):422426*, July 1970.

[5] BROADWELL, P., HARREN, M., and SASTRY, N., "Scrash: A system for generating security crash information," in *Usenix Security Symposium*, 2003.

[6] BRORSSON, M., "SM-prof: a tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs," in *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 178–187, ACM, 1995.

[7] CAIN, H. and LIPASTI, M., "Memory ordering: A value based approach," in *International Symposium on Computer Architecture*, 2004.

[8] CALDER, B., KRINTZ, C., JOHN, S., and AUSTIN, T., "Cache-conscious data placement," *SIGPLAN Not.*, vol. 33, no. 11, pp. 139–149, 1998.

[9] CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., and VLACHOS, E., "Flexible hardware acceleration for instruction-grain program monitoring," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 377–388, IEEE Computer Society, 2008.

[10] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., and IYER, R., "Defeating memory corruption attacks via pointer taintedness detection," in *Proceedings of The International Conference on Dependable Systems and Networks*, 2005.

[11] CHILIMBI, T. M., HILL, M. D., and LARUS, J. R., "Cache-conscious structure layout," *SIGPLAN Not.*, vol. 34, no. 5, pp. 1–12, 1999.

[12] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., and ROSENBLUM, M., "Understanding data lifetime via whole system simulation," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[13] COLLINS, J. D. and TULLSEN, D. M., "Hardware Identification of Cache Conflict Misses," in *Proceedings of the International Symposium on Microarchitecture*, 1999.

[14] CORDER, G. and FOREMAN, D., *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Wiley and Sons, Inc., 2009.

[15] CORLISS, M. L., LEWIS, E. C., and ROTH, A., "DISE: A Programmable Macro Engine for Customizing Applications," in *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, (New York, NY, USA), pp. 362–373, ACM Press, 2003.

[16] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., and HINTON, H., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proc. of the 7th USENIX Security Conference*, (Berkeley CA, USA), pp. 63–78, USENIX Association, Jan. 1998.

[17] CRANDALL, J. R. and CHONG, F. T., "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, (Washington DC, USA), pp. 221–232, IEEE Computer Society, Dec. 2004.

[18] DALTON, M., KANNAN, H., and KOZYRAKIS, C., "Raksha: A flexible information flow architecture for software security," in *International Symposium on Computer Architecture*, 2007.

[19] DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., and CHRYSOS, G. Z., "ProfileMe : Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," in *International Symposium on Microarchitecture*, pp. 292–302, 1997.

[20] DEVIETTI, J., BLUNDELL, C., MARTIN, M. M. K., and ZDANCEWIC, S., "HardBound: Architectural support for spatial safety of the C programming language," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 103–114, 2008.

[21] DRONGOWSKI, P., "Instruction Based Sampling: A New Performance Analysis Technique For AMD Family 10h processors," *AMD Code Analyst Project Report*, 2007.

[22] DUBEY, P., "Recognition, Mining and Synthesis moves computers to the era of Tera," in *Technology@Intel Magizine*, Feb. 2005.

[23] DUBOIS, M., SKEPPSTEDT, J., RICCIULLI, L., RAMAMURTHY, K., and STENSTRÖM, P., "The detection and elimination of useless misses in multiprocessors.," in *ISCA*, pp. 88–97, 1993.

[24] EGGERS, S. and JEREMIASSEN, T., "Eliminating false sharing," in *International Conference on Parallel Processing*, pp. 377–381, Aug. 1991.

[25] GCC TEAM, "GCC Toolchain," *http://gcc.gnu.org*, 2009.

[26] GOLDBERG, A. J. and HENNESSY, J. L., "MTool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 1, pp. 28–40, 1993.

[27] HALL, M., KOGGE, P., KOLLER, J., DINIZ, P., CHAME, J., DRAPER, J., LACOSS, J., GRANACKI, J., BROCKMAN, J., SRIVASTAVA, A., ATHAS, W., FREEH, V., SHIN, J., and PARK, J., "Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture," in *Supercomputing 1999*, November 1999.

[28] HAO, F., KODIALAM, M., and LAKSHMAN, T. V., "Building high accuracy Bloom filters using partitioned hashing," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 277–288, 2007.

[29] HENNESSY, J. and PATTERSON, D., *Computer Architecture: a Quantitative Approach*. Morgan-Kaufmann Publishers, Inc., 2nd ed., 1996.

[30] HILL, M. D., *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, EECS Department, University of California, Berkeley, Nov 1987.

[31] HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., and ROUSSEL, P., "The Microarchitecture of the Pentium® 4 Processor," *Intel Technology Journal*, vol. 5, no. 1, 2001.

[32] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., and HAND, S., "Practical taint based protection using demand emulation," in *European Conference in Computer Systems*, 2006.

[33] HUH, J., CHANG, J., BURGER, D., and SOHI, G. S., "Coherence Decoupling: Making use of incoherence," in *ASPLOS*, pp. 97–106, 2004.

[34] HYPERTRANSPORT CONSORTIUM, "Hypertransport technology-1.0," *http://www.hypertransport.org/*, 2009.

[35] IBM CORPORATION, "IBM Rational Purify," *http://www.ibm.com/software/awdtools/purify/*, 2005.

[36] INTEL CORPORATION, *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, 2002.

[37] JOUPPI, N. P. and OTHERS, "Cacti 4.2," *http://quid.hpl.hp.com:9081/cacti/*, 2006.

[38] KIM, Y. H., HILL, M. D., and WOOD, D. A., "Implementing stack simulation for highly-associative memories," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

[39] KONG, J., ZOU, C., and ZHOU, H., "Improving software security via runtime instruction level taint checking," in *First Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[40] KUEHN, J. T. and SMITH, B. J., "The Horizon supercomputing system: Architecture and software," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 28–34, IEEE Computer Society Press, 1988.

[41] LEPAK, K. and LIPASTI, M., "Temporally silent writes," in *Architectural Support for Programming Languages and Operating Systems*, 2002.

[42] LEVESON, N. and TURNER, C., "An Investigation of the Therac-25 Accidents," in *IEEE Computer*, July 1993.

[43] MARTONOSI, M., GUPTA, A., and ANDERSON, T., "MemSpy: analyzing memory system bottlenecks in programs," in *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 1–12, ACM, 1992.

[44] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., and TRAIGER, I. L., "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[45] MCDONALD, R. G., BURGER, D., and KECKLER, S., "The Design and Implementation of the TRIPS Prototype Chip," *http://www.hotchips.org/archives/hc17*, 2005.

[46] MOUSA, H. and KRINTZ, C., "HPS: Hybrid Profiling Support," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 38–50, IEEE Computer Society, 2005.

[47] NAGPURKAR, P., MOUSA, H., KRINTZ, C., and SHERWOOD, T., "Efficient remote profiling for resource-constrained devices," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 1, pp. 35–66, 2006.

[48] NECULA, G., MCPEAK, S., and WEIMER, W., "CCured: Typesafe retrofitting of legacy code," in *Proceedings of Symposium on Principles of Programming Languages*, 2002.

[49] NETHERCOTE, N. and SEWARD, J., "How to shadow every byte of memory used by a program," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, (New York, NY, USA), pp. 65–74, ACM, 2007.

[50] NEWSOME, J. and SONG, D., "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, (Renton VA, USA), Internet Society, Feb. 2005.

[51] NINTENDO CORPORATION, "The legend of Zelda: Twilight Princess," *http://www.zelda.com/universe/*, 2008.

[52] NINTENDO CORPORATION, "Wii Console," *http://www.nintendo.com/wii*, 2009.

[53] O'REILLY, "Perl security," in *http://search.cpan.org/dist/perl/pod/perlsec.pod*, 2005.

[54] PETRIC, V., SHA, T., and ROTH, A., "RENO: A Rename-Based Instruction Optimizer," in *International Symposium on Computer Architecture*, 2005.

[55] QIN, F., LU, S., and ZHOU, Y., "SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 291–302, IEEE Computer Society, 2005.

[56] QIN, F., WANG, C., LI, Z., KIM, H. S., ZHOU, Y., and WU, Y., "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 135–148, IEEE Computer Society, 2006.

[57] RENAU, J. and OTHERS, "SESC," *http://sesc.sourceforge.net*, 2006.

[58] ROTH, A., "Store vulnerability window: Reexecution filtering for enhanced load optimization," in *International Symposium on Computer Architecture*, 2006.

[59] SASTRY, S. S., BODÍK, R., and SMITH, J. E., "Rapid profiling via stratified sampling," in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 278–289, ACM Press, 2001.

[60] SEWARD, J., "Valgrind, An Open-Source Memory Debugger for x86-GNU/Linux," *http://valgrind.kde.org/*, 2004.

[61] SHANKAR, U., TALWAR, K., FOSTER, J., and WAGNER, D., "Detecting format string vulnerabilities with type qualifiers," in *Usenix Security Symposium*, 2001.

[62] SHETTY, R., KHARBUTLI, M., SOLIHIN, Y., and PRVULOVIC, M., "HeapMon: A helper-thread Approach to programmable, automatic, and low-overhead memory bug detection," *IBM Journal of Research and Development*, vol. 50, no. 2/3, pp. 261–275, 2006.

[63] SHI, T.-P. and DAVIDSON, E. S., "Grouping array layouts to reduce communication and improve locality of parallel programs," in *International Conference on Parallel and Distributed Systems*, 1994.

[64] SHI, W., FRYMAN, J., GU, G., LEE, H.-H., ZHANG, Y., and YANG, J., "InfoShield: A Security Architecture for Protecting Information Usage in Memory," in *International Symposium on High Performance Computer Architecture*, 2006.

[65] SHI, W., LU, C., and LEE, H.-H. S., "Memory centric security architecture," *Transactions on High-Performance Embedded Architectures and Compilers I*, vol. 4050, no. 1, pp. 95–115, 2007.

[66] SKEEL, R., "Roundoff error and patriot missile," in *SIAM News*, July 1992.

[67] SPEC, "Standard Performance Evaluation Corporation Benchmarks," *http://www.spec.org*, 2006.

[68] SUH, G. E., LEE, J. W., ZHANG, D., and DEVADAS, S., "Secure program execution via dynamic information flow tracking," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 85–96, ACM, 2004.

[69] SUN MICROSYSTEMS, "UltraSPARC T2 Supplement," *UltraSPARC Architecture*, 2007.

[70] TASSEY, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing," *NIST Report 02-3*, 2002.

[71] TORRELLAS, J., LAM, M. J., and HENNESSY, J. L., "Shared data placement optimizations to reduce multiprocessor cache misses," in *Proceedings of the International Conference on Parallel Processing*, pp. 266–270, 1990.

[72] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., and AUGUST, D. I., "Rifle: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[73] VALGRIND DEVELOPERS, "The Valgrind Quick Start Guide," *http://valgrind.org/docs/manual/quick-start.html*, 2005.

[74] VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., and PRVULOVIC, M., "FlexiTaint: A Programmable Accelerator for dynamic taint propagation," in *HPCA '08: Proceedings of the 2008 IEEE 14th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 173–184, IEEE Computer Society, 2008.

[75] VENKATARAMANI, G., HUGHES, C. J., KUMAR, S., and PRVULOVIC, M., "Coherence Miss Classification For Performance Debugging in Multi-Core Processors," in

Thirteenth Workshop on Interaction between Compilers and Computer Architecture, (Los Alamitos, CA, USA), IEEE Computer Society Press, 2009.

[76] VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y., and PRVULOVIC, M., "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 273–284, IEEE Computer Society, 2007.

[77] WITCHEL, E., CATES, J., and ASANOVIC, K., "Mondrian memory protection," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, (New York, NY, USA), pp. 304–316, ACM Press, 2002.

[78] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., and GUPTA, A., "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, (New York, NY, USA), pp. 24–36, ACM, 1995.

[79] XU, W., BHATKAR, S., and R.SEKAR, "Taint-enhanced policy enforcement: A practical approach to defeat a range of attacks," in *Usenix Security Symposium*, 2006.

[80] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., and KUMAR, S., "Dynamic tracking of page miss ratio curve for memory management," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[81] ZHOU, P., LIU, W., FEI, L., LU, S., QIN, F., ZHOU, Y., MIDKIFF, S., and TORRELLAS, J., "AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM*

International Symposium on Microarchitecture, (Washington, DC, USA), pp. 269–280, IEEE Computer Society, 2004.

[82] ZHOU, P., QIN, F., LIU, W., ZHOU, Y., and TORRELLAS, J., "iWatcher: Efficient Architectural Support for Software Debugging," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, (Washington, DC, USA), p. 224, IEEE Computer Society, 2004.

[83] ZHOU, Y., ZHOU, P., QIN, F., LIU, W., and TORRELLAS, J., "Efficient and flexible architectural support for dynamic monitoring," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 3–33, 2005.

[84] ZILLES, C. B. and SOHI, G. S., "A programmable co-processor for profiling," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), p. 241, IEEE Computer Society, 2001.